The Substitution Vanishes

Armin Kühnemann and Andreas Maletti*

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

Abstract. Accumulation techniques were invented to transform functional programs, which intensively use append functions (like inefficient list reversal), into more efficient programs, which use accumulating parameters instead (like efficient list reversal). In this paper we present a generalized and automatic accumulation technique that also handles programs operating with unary functions on arbitrary tree structures and employing substitution functions on trees which may replace different designated symbols by different trees. We show that this transformation does not deteriorate the efficiency with respect to call-by-need reduction.

1 Introduction

The sequence of trees in Figure 1 illustrates the stepwise growth of a tree, where in every step in parallel every occurrence of a symbol A (and B, respectively) is substituted by a tree $(D\ A)$ (and $(T\ A\ B\ A)$, respectively).

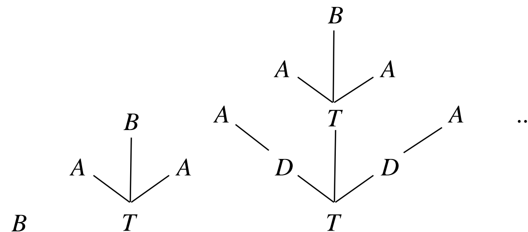


Fig. 1. Stepwise growth of a tree.

In a functional program p_{non} (formulated e.g. in Haskell), this substitution can be defined by a ternary function g that takes the "previous tree" and the two kinds of "fresh branches" $(D\ A)$ and $(T\ A\ B\ A)$ as arguments. Additionally, a unary function f generates as many nested calls of g as the argument of f indicates, where natural numbers are represented by a nullary Z and a unary S, i.e. the *initial expression* $(f\ (S^n\ Z))$ generates n nested calls of g.[1]

* Research of this author supported by DFG under grants GK 334 and KU 1290/2-4.

[1] Since there is only one unary input symbol S for f, the actual parameters $(D\ A)$ and $(T\ A\ B\ A)$ of g are unique. Hence, an alternative version of g could avoid its formal parameters y_1 and y_2 and directly use $(D\ A)$ and $(T\ A\ B\ A)$ in its A- and B-rules, respectively. In a more elaborate example with different unary input symbols for f the actual parameters of g may be different and hence the formal parameters are essential. For convenience we avoided to blow up our example into this direction.

M. Johnson and V. Vene (Eds.): AMAST 2006, LNCS 4019, pp. 173–188, 2006.
© Springer-Verlag Berlin Heidelberg 2006

$$
\begin{aligned}
&f\ Z &&= B\\
&f\ (S\ x_1) &&= g\ (f\ x_1)\ (D\ A)\ (T\ A\ B\ A)\\
&g\ A\ y_1\ y_2 &&= y_1\\
&g\ B\ y_1\ y_2 &&= y_2\\
&g\ (D\ x_1)\ y_1\ y_2 &&= D\ (g\ x_1\ y_1\ y_2)\\
&g\ (T\ x_1\ x_2\ x_3)\ y_1\ y_2 &&= T\ (g\ x_1\ y_1\ y_2)\ (g\ x_2\ y_1\ y_2)\ (g\ x_3\ y_1\ y_2)
\end{aligned}
$$

Unfortunately, p_{non} has cubic time-complexity in the input number n, since g has to process n *intermediate results* with sizes $1^2, 2^2, \ldots, n^2$, respectively (though they are not explicitly constructed in a call-by-need evaluation).

Therefore we would prefer the following program p_{acc} which has linear time-complexity in n to evaluate the (modified) initial expression $(f\ (S^n\ Z)\ A\ B)$:

$$
\begin{aligned}
&f\ Z\ y_1\ y_2 &&= y_2\\
&f\ (S\ x_1)\ y_1\ y_2 &&= f\ x_1\ (D\ y_1)\ (T\ y_1\ y_2\ y_1)
\end{aligned}
$$

Since p_{acc} uses its second and third argument to accumulate step by step a "D-branch" and an output tree, respectively, we call p_{acc} an *accumulative* program, whereas we call p_{non} *non-accumulative*. Techniques which transform non-accumulative into accumulative programs are called *accumulation*.

In the case that substitutions on tree structures are restricted to append functions on list structures, there is a long history of research on accumulation: Already in [6] it is shown in the context of transforming programs into iterative (tail-recursive) form, how non-accumulative programs can be transformed (non-automatically) into their accumulative versions. In [3,18] a similar technique for linear recursive functions is presented. Other non-automatic realizations of accumulation are given e.g. in [4,14]. Finally, the transformation of [25] is fully automatic and is accompanied by an efficiency analysis. The crucial laws, on which the transformation of [25] is based, can be found in our paper in a similar form. All mentioned techniques essentially rely on the properties of the monoid of lists with append. This fact is detailed in [5].

Our automatic transformation technique is more general in two aspects: we consider (i) arbitrary tree structures (instead of lists) as input and output, and (ii) substitutions on trees (instead of append) which additionally 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 [25] the only example program involving a non-unary function could not be optimized. Hence the scope of our technique includes the unary functions in the examples of [25] (in particular, inefficient list reversal). Moreover, restricting recursive calls of the unary functions to be primitive-recursive will guarantee that in contrast to [25] no substitutions appear in transformed programs anymore. Our efficiency result is based on exactly this fact.

For this purpose, we view functional programs like p_{non} as special 2-*modular tree transducers* [10]. Every function in module 1 (like f in p_{non}) is unary and is defined by a case analysis on the root symbol c of its argument t. The right-hand side of the equation for f and c may contain (primitive-recursive) calls of functions in module 1 on subtrees of t and arbitrary calls of the (only) function in module 2. The function in module 2 (like g in p_{non}) is a *substitution function*,

i.e. designated nullary *substitution constructors* are replaced by parameters and other constructors are left unchanged. In [16] it was shown, how such programs can be transformed into *macro tree transducers* [8,9], in which every function (like f in p_{acc}) may have arbitrary rank and is defined by a case analysis on the root symbol c of its first argument t. The right-hand side of the equation for f and c may contain (nested) recursive function calls on subtrees of t.

The accumulation technique of [16] is divided into three indirect transformation steps which are mainly based on a composition result [9,15] for *top-down tree transducers* [19,21] with macro tree transducers and on the associativity of substitution functions. Although the resulting programs avoid substitution functions, they are not always more efficient than the original programs. In [16] and in the present paper the efficiency is measured in the number of performed call-by-need reduction steps. This point of view, which neglects the actual complexity of rule applications, is also taken in, e.g., [20,23,24].

In [16] also the reverse transformation is presented. Both transformations together induce that the classes of macro tree transducers and of the special 2-modular tree transducers have the same computational power. Although the reverse transformation deteriorates in general the efficiency, it also has practical relevance: In [12] it is extended to a *deaccumulation* technique which is useful to improve the automatic verification of functional (and even imperative) programs.

In [17] the deficiencies of the accumulation technique in [16] were solved by presenting a direct transformation which additionally employs let-expressions to avoid causes of inefficiency. Moreover, it was shown in [17] that the transformation does not deteriorate the efficiency. To this end, a call-by-need reduction on term graphs was defined and compared for the original and resulting program. The efficiency result is based on the fact that the number of applications of functions in module 1 of the original program equals the number of function applications in the resulting program. Hence, the applications of the substitution function in module 2 of the original program are saved!

We simplify the presentation of [17] by avoiding an explicit call-by-need reduction and by adopting a technique of [20,23,24], where function applications (in [23,24] for (compositions of) macro tree transducers) additionally produce special “ticking symbols” in order to make the number of performed (call-by-name) reduction steps visible in the output. Instead of a call-by-need reduction relation on term graphs which (implicitly) uses sharing to avoid that unevaluated function arguments are copied, we use a nondeterministic reduction relation on expressions with an explicit denotation for sharing (cf., e.g., [2,1]). Unfortunately, this *explicit sharing* does not prevent that our nondeterministic reduction relation creates a shared subexpression e such that e contains function applications and e is not relevant in the overall expression (in the sense that call-by-need reduction would delete all references to e). To avoid that ticking symbols which are generated by e are counted (and thus reduction steps needed to evaluate e), we additionally use a *counting function* which takes care of such nonrelevant subexpressions. Hence the concepts of *explicit sharing* and the *counting function* provide a new technique to count call-by-need reduction steps.

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 use the sets $X = \{x_0, x_1, x_2, x_3, \ldots\}$, $Y = \{y_1, y_2, y_3, \ldots\}$, and $Z = \{z_1, z_2, z_3, \ldots\}$ of *variables*. For every $n \in \mathbb{N}$, let $X_n = \{x_1, \ldots, x_n\}$, $Y_n = \{y_1, \ldots, y_n\}$, and $Z_n = \{z_1, \ldots, z_n\}$. Let $\Rightarrow$ be a binary relation on a set K and $n \in \mathbb{N}$. Then, $\Rightarrow^n$ denotes the n-fold composition and $\Rightarrow^*$ the transitive, reflexive closure of $\Rightarrow$. If $k \Rightarrow^* k'$ for $k, k' \in K$ and 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 $(C, rank)$ where C is a finite set and $rank$ is a mapping which associates with every symbol $c \in C$ a natural number called the *rank* of c. We simply write C instead of $(C, rank)$ and assume $rank$ as implicitly given. The set of symbols of C with rank n is denoted by $C^{(n)}$ and if $c \in C^{(n)}$, we also use the notation $c^{(n)}$. The set of *trees* (or *terms*) *over C indexed by* (*a set of variables*) U, denoted by $T_C(U)$, is the smallest subset $T \subseteq (C \cup U \cup \{(,)\})^*$ such that $U \subseteq T$ and for every $c \in C^{(n)}$ with $n \in \mathbb{N}$ and $t_1, \ldots, t_n \in T$: $(c\ t_1 \ldots t_n) \in T$. If $c \in C^{(0)}$, we write just c instead of (c). The set $T_C(\emptyset)$ is abbreviated by T_C. If R is the set of rules of a term rewriting system, then $\Rightarrow_R$ denotes the (nondeterministic) reduction relation induced by R. If there is at most one occurrence of a variable v in a term t, then we call t *linear* in v.

For a term t, pairwise distinct variables $v_1, \ldots, v_n$, and terms $t_1, \ldots, t_n$, we denote by $t[v_1/t_1, \ldots, v_n/t_n]$ the term that is obtained from t by substituting for every $i \in [n]$ every occurrence of v_i in t by t_i. We abbreviate $[v_1/t_1, \ldots, v_n/t_n]$ by $[v_i/t_i]$, if the involved variables and terms are clear from the context. We use a linear, "substitution-like" notation for term graphs to express the sharing of subgraphs: $e[z_1 \rightsquigarrow e_1, \ldots, z_n \rightsquigarrow e_n]$ denotes a term graph, in which for every occurrence of z_i in the subgraph denoted by e there is a directed edge from the direct ancestor node of z_i to the root node of the subgraph denoted by e_i.

For the rest of the paper, let $n \in \mathbb{N}_+$.

Definition 1 Let C be a ranked alphabet and $U \in \{Z_n, \emptyset\}$. The set $E_{C,n}(U)$ of *C-expressions with sharing* (*and free variables of U*) is defined by:

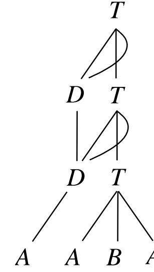
- For every $z_j \in U$: $z_j \in E_{C,n}(U)$.
- For every $c \in C^{(k)}$ and $e_1, \ldots, e_k \in E_{C,n}(U)$: $(c\ e_1 \ldots e_k) \in E_{C,n}(U)$.
- For every $e_1, \ldots, e_n \in E_{C,n}(U)$ and $e \in E_{C,n}(Z_n)$: $e[z_1 \rightsquigarrow e_1, \ldots, z_n \rightsquigarrow e_n] \in E_{C,n}(U)$.

The set $E_{C,n}$ of *C-expressions with sharing* is defined as $E_{C,n}(\emptyset)$. □

Note that a C-expression with sharing $e \in E_{C,n}(U)$ can be considered as a tree $e \in T_{C'}(U)$ where $C' = C \cup \{(\cdot[z_1 \rightsquigarrow \cdot, \ldots, z_n \rightsquigarrow \cdot])^{(n+1)}\}$ is the ranked alphabet obtained from C by adding a new $(n+1)$-ary symbol. Thus we can employ the notions and notations, which we introduced for trees, also for C-expressions.

Example 2 Let $C = \{A^{(0)}, B^{(0)}, D^{(1)}, T^{(3)}\}$. Then,

$$\begin{array}{ll} (T\ z_1\ z_2\ z_1) & \in E_{C,2}(Z_2), \\ (T\ z_1\ z_2\ z_1)[z_1 \rightsquigarrow (D\ z_1), z_2 \rightsquigarrow (T\ z_1\ z_2\ z_1)] & \in E_{C,2}(Z_2), \\ (T\ z_1\ z_2\ z_1)[z_1 \rightsquigarrow (D\ z_1), z_2 \rightsquigarrow (T\ z_1\ z_2\ z_1)] & \\ \qquad [z_1 \rightsquigarrow (D\ A), z_2 \rightsquigarrow (T\ A\ B\ A)] & \in E_{C,2}, \end{array}$$



and the latter represents the depicted term graph. □

If $e_1, \ldots, e_n$ are clear from the context, then we abbreviate an expression of the form $e[z_1 \rightsquigarrow e_1, \ldots, z_n \rightsquigarrow e_n]$ by $e[z_i \rightsquigarrow e_i]$.

In the following we define a function on C-expressions with sharing, which constructs trees by unfolding all sharings.

Definition 3 Let C be a ranked alphabet and $U \in \{Z_n, \emptyset\}$. The function $\underline{tree}$: $E_{C,n}(U) \rightarrow T_C(U)$ is defined as follows:

- For every $z_j \in U$: $\underline{tree}(z_j) = z_j$.
- For every $c \in C^{(k)}$ and $e_1, \ldots, e_k \in E_{C,n}(U)$: $\underline{tree}(c\ e_1 \ldots e_k) = (c\ \underline{tree}(e_1) \ldots \underline{tree}(e_k))$.
- For every $e_1, \ldots, e_n \in E_{C,n}(U)$ and $e \in E_{C,n}(Z_n)$: $\underline{tree}(e[z_i \rightsquigarrow e_i]) = \underline{tree}(e)[z_i/\underline{tree}(e_i)]$. □

We call $z_i \in Z_n$ a *free occurrence in* $e \in E_{C,n}(Z_n)$, if z_i occurs in $\underline{tree}(e)$. Note that this clarifies the scope of a sharing. The scope of $z_1, \ldots, z_n$ in an expression $e[z_i \rightsquigarrow e_i]$ is limited to the free occurrences of $z_1, \ldots, z_n$ in e.

3 Nonaccumulating and Accumulating Tree Transducers

First we define nonaccumulating tree transducers as functional source language. Nonaccumulating tree transducers are special 2-modular tree transducers [10,16].

Definition 4 An *n-nonaccumulating tree transducer* (for short *n-ntt*) is a tuple $M = (F, Sub, C, \Pi, R_1, R_2, r_{in})$, where

- F is a ranked alphabet (of *function symbols*) with $F = F^{(1)}$,
- $Sub = \{sub^{(n+1)}\}$ (*substitution function*),
- C is a ranked alphabet (of *constructors*),
- $\Pi = \{\Pi_1, \ldots, \Pi_n\} \subseteq C^{(0)}$ with $card(\Pi) = n$ (the *substitution constructors*)

such that F, Sub, and C are pairwise disjoint,

- R_1 is a set of *rules* such that for every $f \in F$ and $c \in C^{(k)}$ there is exactly one rule $\quad f\ (c\ x_1 \ldots x_k) = rhs_{R_1,f,c}$
  with $rhs_{R_1,f,c} \in RHS(F, Sub, C, X_k)$, where for every $k \in I\!N$, $RHS(F, Sub, C, X_k)$ is the smallest set RHS such that
  - for every $f \in F$ and $i \in [k]$: $(f\ x_i) \in RHS$,
  - for every $r_0, \ldots, r_n \in RHS$: $(sub\ r_0 \ldots r_n) \in RHS$, and
  - for every $c \in C^{(l)}$ and $r_1, \ldots, r_l \in RHS$: $(c\ r_1 \ldots r_l) \in RHS$,
- R_2 is a set of *rules* such that

- for every $j \in [n]$ there is the rule $\quad sub\ \Pi_j\ y_1 \dots y_n = y_j$
- and for every $c \in (C - \Pi)^{(k)}$ there is the rule
  $sub\ (c\ x_1 \dots x_k)\ y_1 \dots y_n = c\ (sub\ x_1\ y_1 \dots y_n) \dots (sub\ x_k\ y_1 \dots y_n)$,

– $r_{in} \in RHS(F, Sub, C, X_1)$ is the *initial right-hand side*. □

Since every function is defined by recursion on its first argument (i.e., the only argument in case of F-functions), this argument is called *recursion argument*. The other arguments are called *context arguments*. The set *RHS* formalizes the description of right-hand sides found in the introduction. The initial right-hand side r_{in} serves as *call pattern* for the n-ntt, where x_1 acts as a placeholder for the actual input tree. Note that the concept of n-ntts (with one substitution function of rank $n + 1$) can easily be generalized to a model with several substitution functions. This, however, does not increase the computational power.

In the following examples we will avoid rules, which are never used.

Example 5 $M_{non} = (F, Sub, C, \Pi, R_1, R_2, r_{in})$ is a 2-ntt where $F = \{f\}$, $Sub = \{g^{(3)}\}$, $C = \{S^{(1)}, Z^{(0)}, A^{(0)}, B^{(0)}, D^{(1)}, T^{(3)}\}$, $\Pi_1 = A$, $\Pi_2 = B$, R_1 and R_2 contain the f-rules and g-rules, respectively, of p_{non}, and $r_{in} = (f\ x_1)$. □

Now we present n-ntts with sharings as abstractions for our functional source programs. In contrast to functional programs, where in a call-by-need reduction the sharing of expressions which are bound to variables of rules is performed implicitly, in n-ntts with sharings the sharing is performed explicitly, whenever there is the risk to copy unevaluated expressions (cf., e.g., [2,1]). This concerns only the context arguments of substitution functions (since other arguments are not copied or are constructor trees). To denote explicit sharing in a right-hand side of a rule or in a sentential form, we also use expressions with sharing. Thus, because of possibly nested substitution functions during an evaluation, also expressions with sharing may occur in the recursion argument of substitution functions. Hence they must be handled by a special rule. Actually, n-ntts with sharings could be considered as special "2-modular tree-to-graph transducers". See [10,11] for the concepts of modular tree transducers and top-down tree-to-graph transducers, respectively. Note that the additional sharing mechanism does not change the computational power of n-ntts, but may improve efficiency.

Definition 6 An *n-nonaccumulating tree transducer with sharings* (for short *n-sntt*) is a tuple $M = (F, Sub, C, \Pi, R_1, R_2, r_{in})$, where

– F, Sub, C, Π, R_1, and r_{in} are defined as in Definition 4,
– R_2 is a set of *rules* such that
  - for every $j \in [n]$ there is the rule $\quad sub\ \Pi_j\ y_1 \dots y_n = y_j$,
  - for every $c \in (C - \Pi)^{(k)}$ there is the rule[2]
    $sub\ (c\ x_1 \dots x_k)\ y_1 \dots y_n$
    $= (c\ (sub\ x_1\ z_1 \dots z_n) \dots (sub\ x_k\ z_1 \dots z_n))[z_i \rightsquigarrow y_i]$,
  - and there is the rule[3]
    $sub\ x_0[z_i \rightsquigarrow x_i]\ y_1 \dots y_n = x_0[z_i \rightsquigarrow (sub\ x_i\ z_1 \dots z_n)][z_i \rightsquigarrow y_i]$. □

[2] If c is nullary or unary, then the explicit sharing will be avoided in examples.
[3] If $n = 1$, then the explicit sharing $[z_i \rightsquigarrow y_i]$ could be avoided.

Note that in the last rule in the previous definition *sub* walks into $x_1, \ldots, x_n$, but not into x_0. This is due to the fact that every instantiation of $x_0[z_i \rightsquigarrow x_i]$ was generated by an inner occurrence of *sub* which already handled the substitution constructors in x_0. Moreover, it is easily seen that the inner occurrence of *sub* does not introduce substitution constructors in x_0 because only calls of the form $(sub\ x_i\ z_1 \ldots z_n)$ can occur in x_0. We further note that an application of the last rule represents a short cut, since a call-by-need reduction on term graphs would (i) walk stepwise through the expression bound to x_0 and would (ii) end up with different occurrences of *sub* at different occurrences of a z_i (thus performing several runs on the expression bound to x_i).

Example 7 $\tilde{M}_{non} = (F, Sub, C, \Pi, R_1, R_2, r_{in})$ is a 2-sntt, where $F = \{f\}$, $Sub = \{g^{(3)}\}$, $C = \{S^{(1)}, Z^{(0)}, A^{(0)}, B^{(0)}, D^{(1)}, T^{(3)}\}$, $\Pi_1 = A$, $\Pi_2 = B$, R_1 contains the f-rules of p_{non} and R_2 contains rules

$$\begin{array}{ll} g\ A\ y_1\ y_2 & = y_1 \\ g\ B\ y_1\ y_2 & = y_2 \\ g\ (D\ x_1)\ y_1\ y_2 & = D\ (g\ x_1\ y_1\ y_2) \\ g\ (T\ x_1\ x_2\ x_3)\ y_1\ y_2 & = (T\ (g\ x_1\ z_1\ z_2)\ (g\ x_2\ z_1\ z_2)\ (g\ x_3\ z_1\ z_2))[z_i \rightsquigarrow y_i] \\ g\ x_0[z_i \rightsquigarrow x_i]\ y_1\ y_2 & = x_0[z_i \rightsquigarrow (g\ x_i\ z_1\ z_2)][z_i \rightsquigarrow y_i], \end{array}$$

and $r_{in} = (f\ x_1)$. Let $R = R_1 \cup R_2$. Then,

$$\begin{array}{lll} f\ (S^3\ Z) & \Rightarrow_R^5 & g\ (g\ (T\ A\ B\ A)\ (D\ A)\ (T\ A\ B\ A))\ (D\ A)\ (T\ A\ B\ A) \\ & \Rightarrow_R^4 & g\ (T\ z_1\ z_2\ z_1)[z_1 \rightsquigarrow (D\ A), z_2 \rightsquigarrow (T\ A\ B\ A)]\ (D\ A)\ (T\ A\ B\ A) \\ & \Rightarrow_R & (T\ z_1\ z_2\ z_1)[z_1 \rightsquigarrow (g\ (D\ A)\ z_1\ z_2), z_2 \rightsquigarrow (g\ (T\ A\ B\ A)\ z_1\ z_2)] \\ & & \quad [z_1 \rightsquigarrow (D\ A), z_2 \rightsquigarrow (T\ A\ B\ A)] \\ & \Rightarrow_R^6 & (T\ z_1\ z_2\ z_1)[z_1 \rightsquigarrow (D\ z_1), z_2 \rightsquigarrow (T\ z_1\ z_2\ z_1)[z_1 \rightsquigarrow z_1, z_2 \rightsquigarrow z_2]] \\ & & \quad [z_1 \rightsquigarrow (D\ A), z_2 \rightsquigarrow (T\ A\ B\ A)]. \end{array}$$

□

Our main transformation will deliver accumulating tree transducers with sharings, which could be considered as special "macro tree-to-graph transducers". See [9,11] for the concepts of macro tree transducers and top-down tree-to-graph transducers, respectively.

Definition 8 An *n-accumulating tree transducer with sharings* (for short *n-satt*) is a tuple $M = (F, C, R, r_{in})$, where

- F is a ranked alphabet (of *function symbols*) with $F = F^{(n+1)}$,
- C is a ranked alphabet (of *constructors*), such that F and C are disjoint,
- R is a set of *rules* such that for every $f \in F$ and $c \in C^{(k)}$ there is exactly one rule $f\ (c\ x_1 \ldots x_k)\ y_1 \ldots y_n = rhs_{R,f,c}$
  with $rhs_{R,f,c} \in RHS'(F, C, X_k, Y_n)$, where for every $j \in [n]$, the right-hand side $rhs_{R,f,c}$ is linear in y_j, and for every $k \in I\!N$ and $U \in \{Y_n, Z_n, \emptyset\}$, the set $RHS'(F, C, X_k, U)$ is the smallest set such that
  - for every $f \in F$, $i \in [k]$, and $r_1, \ldots, r_n \in RHS'(F, C, X_k, U)$: $(f\ x_i\ r_1 \ldots r_n) \in RHS'(F, C, X_k, U)$,
  - for every $c \in C^{(l)}$ and $r_1, \ldots, r_l \in RHS'(F, C, X_k, U)$: $(c\ r_1 \ldots r_l) \in RHS'(F, C, X_k, U)$,

- for every $r_1, \ldots, r_n \in RHS'(F, C, X_k, U)$ and $r_0 \in RHS'(F, C, X_k, Z_n)$: $r_0[z_i \rightsquigarrow r_i] \in RHS'(F, C, X_k, U)$, and
- for every $u \in U$: $u \in RHS'(F, C, X_k, U)$,

– $r_{in} \in RHS'(F, C, X_1, \emptyset)$ is the *initial right-hand side*. □

The linearity condition in the previous definition will be called *context-linearity.* Note that it guarantees that no unevaluated subexpressions are copied in a nondeterministic reduction relation. This fact will be needed in Subsections 4.3 and 5.3, where n-satts are realized by functional programs under call-by-need evaluation, such that the number of performed reduction steps is equal.

Example 9 $M_{acc} = (F', C, R, r'_{in})$ is a 2-satt, where $F' = \{f^{(3)}\}$, $C = \{S^{(1)}, Z^{(0)}, A^{(0)}, B^{(0)}, D^{(1)}, T^{(3)}\}$, R contains the rules

$$\begin{aligned} f\ Z\ y_1\ y_2 &= y_2 \\ f\ (S\ x_1)\ y_1\ y_2 &= (f\ x_1\ (D\ z_1)\ (T\ z_1\ z_2\ z_1))[z_i \rightsquigarrow y_i] \end{aligned}$$

and $r'_{in} = (f\ x_1\ A\ B)$. Then,

$$\begin{aligned} f\ (S^3\ Z)\ A\ B &\Rightarrow_R\ (f\ (S^2\ Z)\ (D\ z_1)\ (T\ z_1\ z_2\ z_1))[z_1 \rightsquigarrow A, z_2 \rightsquigarrow B] \\ &\Rightarrow_R\ (f\ (S\ Z)\ (D\ z_1)\ (T\ z_1\ z_2\ z_1)) \\ &\qquad [z_1 \rightsquigarrow (D\ z_1), z_2 \rightsquigarrow (T\ z_1\ z_2\ z_1)][z_1 \rightsquigarrow A, z_2 \rightsquigarrow B] \\ &\Rightarrow_R^2\ (T\ z_1\ z_2\ z_1)[z_1 \rightsquigarrow (D\ z_1), z_2 \rightsquigarrow (T\ z_1\ z_2\ z_1)] \\ &\qquad [z_1 \rightsquigarrow (D\ z_1), z_2 \rightsquigarrow (T\ z_1\ z_2\ z_1)][z_1 \rightsquigarrow A, z_2 \rightsquigarrow B]. \end{aligned}$$

□

For every n-sntt $M = (F, Sub, C, \Pi, R_1, R_2, r_{in})$ with $R = R_1 \cup R_2$ and for every n-satt $M = (F, C, R, r_{in})$, $\Rightarrow_R$ is locally confluent, because there are no critical pairs. Similarly to modular tree transducers [10] and macro tree transducers [9], $\Rightarrow_R$ is also terminating, since every rule application to a function symbol with its recursion argument t delivers only (i) new function symbols with subtrees of t as recursion arguments or (in the case of an n-sntt) (ii) occurrences of the substitution function which does not call any other function and also "strictly walks down" on its recursion argument. Thus, for every $t \in T_C$, $nf(\Rightarrow_R, r_{in}[x_1/t])$ exists. Moreover, there are no function symbols in this normal form, because all functions are exhaustively defined on their possible recursion arguments (in particular on all outputs of functions which are nested in their recursion arguments). Hence, the normal form is a C-expression with sharing.

Definition 10 Let $M = (F, Sub, C, \Pi, R_1, R_2, r_{in})$ be an n-sntt with $R = R_1 \cup R_2$ or let $M = (F, C, R, r_{in})$ be an n-satt. The *tree transformation computed by* M is the function $\tau_M : T_C \to T_C$, which is for every $t \in T_C$ defined by $\tau_M(t) = \underline{tree}(nf(\Rightarrow_R, r_{in}[x_1/t]))$. □

4 Accumulation Technique

Our transformation technique consists of three steps: (i) a preprocessing step which abstracts n-ntts into n-sntts, (ii) the main transformation on the level of tree transducers with sharings (transforming n-sntts into n-satts), and (iii) a postprocessing step which realizes n-satts as functional programs. Since the pre-

and postprocessing steps are relatively simple compared to the main transformation, they will be presented only informally.

4.1 Preprocessing

Explicit sharings which were introduced in the previous section, are added. More exactly, the *sub*-rules of Definition 4 are replaced by those of Definition 6. Note that this preprocessing step simplifies our efficiency considerations; the main transformation could also take n-ntts as inputs.

4.2 Main Transformation

The main transformation processes an n-sntt M and yields an n-satt M'. The construction introduces a new $(n+1)$-ary function symbol f for every function symbol f of M. The context arguments of the new f shall store replacements for the substitution constructors. Intuitively speaking, a call like $(f\ t\ e_1 \ldots e_n)$ should evaluate (using M') to the result of $(sub\ (f\ t)\ e_1 \ldots e_n)$ (evaluated using M). Thereby the intermediate result that is produced by the call $(f\ t)$ is avoided. The formalization of this intuitive relation can be found in Lemma 13. The construction uses an auxiliary function $\underline{sub}$ to transform right-hand sides of rules thereby evaluating substitutions at compile time.

Definition 11 Let $M = (F, Sub, C, \Pi, R_1, R_2, r_{in})$ be an n-sntt. First, we define the set $\overline{R_2}$ of *transformation rules* which contains

- for every $j \in [n]$ a rule $\underline{sub}\ \Pi_j\ y_1 \ldots y_n = y_j$,
- for every $c \in (C - \Pi)^{(k)}$ a rule[4]
  $\underline{sub}\ (c\ x_1 \ldots x_k)\ y_1 \ldots y_n$
  $= (c\ (\underline{sub}\ x_1\ z_1 \ldots z_n) \ldots (\underline{sub}\ x_k\ z_1 \ldots z_n))[z_i \leadsto y_i]$,
- for every $f \in F$ a rule $\underline{sub}\ (f\ x_0)\ y_1 \ldots y_n = f\ x_0\ y_1 \ldots y_n$,
- and the rule[5]
  $\underline{sub}\ (sub\ x_0\ x_1 \ldots x_n)\ y_1 \ldots y_n$
  $= (\underline{sub}\ x_0\ (\underline{sub}\ x_1\ z_1 \ldots z_n) \ldots (\underline{sub}\ x_n\ z_1 \ldots z_n))[z_i \leadsto y_i]$.

Then, the *n-satt constructed from M by accumulation* is defined as $acc(M) = (acc(F), C, acc(R_1), acc(r_{in}))$, where

- $acc(F) = \{f^{(n+1)} \mid f \in F\}$,
- $acc(R_1)$ contains for every $f \in acc(F)$ and $c \in C^{(k)}$ the rule
  $f\ (c\ x_1 \ldots x_k)\ y_1 \ldots y_n = nf(\Rightarrow_{\overline{R_2}}, \underline{sub}\ rhs_{R_1,f,c}\ y_1 \ldots y_n)$,
- $acc(r_{in}) = nf(\Rightarrow_{\overline{R_2}}, \underline{sub}\ r_{in}\ \Pi_1 \ldots \Pi_n)$. □

It can be shown easily that $acc(M)$ is in fact a well-defined n-satt. In particular, the context-linearity is induced by the fact that the $\underline{sub}$-rules do not copy variables. Given the above intuition, the rules for $\underline{sub}$ should be straightforward: The first rule avoids the explicit construction of Π_j-symbols. The second rule is standard and the third rule encodes our intuition. Finally, the fourth rule

[4] If c is nullary or unary, then the explicit sharing will be avoided in examples.
[5] If $n = 1$, then the explicit sharing could be avoided.

represents the "associativity" of substitutions. Note the similarity of these rules to the laws (1), (2), (∗), and (3), respectively, of [25]. In the second and fourth rule we use explicit sharings in order to avoid that occurrences of F-functions are copied and thus are executed more than once in the transformed program.

Example 12 Let $\tilde{M}_{non} = (F, Sub, C, \Pi, R_1, R_2, r_{in})$ be the 2-sntt from Example 7. Then, the set $\overline{R_2}$ contains the rules

$$
\begin{aligned}
\underline{g}\ A\ y_1\ y_2 &= y_1 \\
\underline{g}\ B\ y_1\ y_2 &= y_2 \\
\underline{g}\ (D\ x_1)\ y_1\ y_2 &= D\ (\underline{g}\ x_1\ y_1\ y_2) \\
\underline{g}\ (T\ x_1\ x_2\ x_3)\ y_1\ y_2 &= (T\ (\underline{g}\ x_1\ z_1\ z_2)\ (\underline{g}\ x_2\ z_1\ z_2)\ (\underline{g}\ x_3\ z_1\ z_2))[z_i \rightsquigarrow y_i] \\
\underline{g}\ (f\ x_0)\ y_1\ y_2 &= f\ x_0\ y_1\ y_2 \\
\underline{g}\ (g\ x_0\ x_1\ x_2)\ y_1\ y_2 &= (\underline{g}\ x_0\ (\underline{g}\ x_1\ z_1\ z_2)\ (\underline{g}\ x_2\ z_1\ z_2))[z_i \rightsquigarrow y_i]
\end{aligned}
$$

and the 2-satt constructed from $\tilde{M}_{non}$ by accumulation is defined as $\tilde{M}_{acc} = acc(\tilde{M}_{non}) = (acc(F), C, acc(R_1), acc(r_{in}))$, where $acc(F) = \{f^{(3)}\}$, $acc(R_1)$ contains the following rules with underlined left- and right-hand sides

$$
\begin{aligned}
&\underline{f\ Z\ y_1\ y_2} = nf(\Rightarrow_{\overline{R_2}}, \underline{g}\ B\ y_1\ y_2) = \underline{y_2}, \\
&\underline{f\ (S\ x_1)\ y_1\ y_2} = nf(\Rightarrow_{\overline{R_2}}, \underline{g}\ (g\ (f\ x_1)\ (D\ A)\ (T\ A\ B\ A))\ y_1\ y_2) \\
&= nf(\Rightarrow_{\overline{R_2}}, (\underline{g}\ (f\ x_1)\ (\underline{g}\ (D\ A)\ z_1\ z_2)\ (\underline{g}\ (T\ A\ B\ A)\ z_1\ z_2))[z_i \rightsquigarrow y_i]) \\
&= \underline{(f\ x_1\ (D\ z_1)\ (T\ z_1\ z_2\ z_1)[z_i \rightsquigarrow z_i])[z_i \rightsquigarrow y_i]}
\end{aligned}
$$

and $acc(r_{in}) = nf(\Rightarrow_{\overline{R_2}}, \underline{g}\ (f\ x_1)\ A\ B) = (f\ x_1\ A\ B)$. □

The correctness proof[6] of the main transformation is based on the following lemma which formalizes our intuition from the beginning of this subsection.

Lemma 13 Let $M = (F, Sub, C, \Pi, R_1, R_2, r_{in})$ be an n-sntt and $acc(M) = (acc(F), C, acc(R_1), acc(r_{in}))$. For every $f \in F$ and $t \in T_C$:

$$\underline{tree}(nf(\Rightarrow_{R_1 \cup R_2}, sub\ (f\ t)\ z_1 \ldots z_n)) = \underline{tree}(nf(\Rightarrow_{acc(R_1)}, f\ t\ z_1 \ldots z_n)). \quad \square$$

Theorem 14 Let M be an n-sntt. Then, $\tau_M = \tau_{acc(M)}$. □

4.3 Postprocessing

Finally, an n-satt resulting from the main transformation is translated into a functional program by replacing in the right-hand sides of rules and in the initial right-hand side the explicit sharings with let-expressions. More exactly, an expression of the form

$$r[z_1 \rightsquigarrow r_1, \ldots, z_n \rightsquigarrow r_n] \qquad \text{is replaced by} \qquad let\ \{v_1 = \overline{r_1}; \ldots; v_n = \overline{r_n}\}\ in\ \overline{r},$$

where $v_1, \ldots, v_n$ are fresh variables (which can be obtained using tree-structured addresses in the translation process) and $\overline{r_1}, \ldots, \overline{r_n}, \overline{r}$ result from recursively replacing explicit sharings in $r_1, \ldots, r_n, r$, respectively, and additionally using $v_1, \ldots, v_n$ instead of the free occurrences of $z_1, \ldots, z_n$ in r.

[6] Available at www.orchid.inf.tu-dresden.de/gdp/conferences/amast06.shtml

Example 15 Let $\tilde{M}_{acc}$ be the 2-satt of Example 12. Postprocessing translates $f\ (S\ x_1)\ y_1\ y_2 = (f\ x_1\ (D\ z_1)\ (T\ z_1\ z_2\ z_1)[z_i \rightsquigarrow z_i])[z_i \rightsquigarrow y_i]$ into the rule

$$f\ (S\ x_1)\ y_1\ y_2 = let\ \{v_1 = y_1; v_2 = y_2\}\\ in\ f\ x_1\ (D\ v_1)\ (let\ \{v_{11} = v_1; v_{12} = v_2\}\ in\ (T\ v_{11}\ v_{12}\ v_{11})). \square$$

A more elaborate translation could simplify (or even avoid) some let-expressions, e.g. if z_j does not occur or occurs only once freely in r or if $r_j = z_j$ or $r_j = y_j$ (i.e. we have $z_j \rightsquigarrow z_j$ or $z_j \rightsquigarrow y_j$). For the case $r_j = y_j$ note that the resulting program will be again treated call-by-need, and hence y_j is shared implicitly.

Example 16 Instead of constructing the rule as in Example 15, the following rule can be used (cf. also the introduction):

$$f\ (S\ x_1)\ y_1\ y_2 = f\ x_1\ (D\ y_1)\ (T\ y_1\ y_2\ y_1) \qquad \square$$

5 Efficiency Non-deterioration by Accumulation

Our aim is to show the efficiency non-deterioration for call-by-need reduction. Unfortunately, it is technically difficult to formally compare the number of steps caused by deterministic reduction relations (cf. e.g. [17]). Hence we will base our comparison on the nondeterministic reduction relations for n-sntts and n-satts.

Therefore we first present a mechanism such that the number of call-by-need reduction steps caused by the R_1-rules of an n-ntt M equals the number of "relevant" nondeterministic reduction steps caused by the R_1-rules of the corresponding n-sntt $\tilde{M}$: In both reduction relations the copying of unevaluated applications of F-functions is avoided (by implicit and explicit sharing, respectively). But, whereas the deletion of a useless unevaluated application of an F-function is performed automatically in a call-by-need reduction, the nondeterministic reduction relation for $\tilde{M}$ either simply evaluates such an application and later moves the result into a subexpression e_i of an expression of the form $e[z_i \rightsquigarrow e_i]$ or, vice versa, the reduction relation for $\tilde{M}$ first moves it into an e_i of an expression $e[z_i \rightsquigarrow e_i]$, where it is evaluated later. In both situations the normal form of e will not contain a free occurrence of z_i, but nevertheless the useless evaluation is performed! In order to consider only the relevant R_1-reduction steps, in our mechanism (i) every application of an R_1-rule will additionally generate a special symbol $\circ$ and (ii) in the normal form only those $\circ$-symbols are counted by a function $\underline{step}$, which do not occur in a subexpression e_i of an expression $e[z_i \rightsquigarrow e_i]$, where z_i does not occur freely in e.

Then we use the same counting mechanism for the n-satt $acc(\tilde{M})$ in order to prove that the number of relevant R_1-reduction steps of $\tilde{M}$ equals the number of relevant reduction steps of $acc(\tilde{M})$. Together with a final argumentation that the postprocessing phase does not change the number of reduction steps, we obtain the desired efficiency result. Note that our comparison procedure does not consider the R_2-reduction steps of M or $\tilde{M}$, which do not occur in $acc(\tilde{M})$ and hence are saved by accumulation!

In the following we assume that $\circ$ is a new unary symbol and for every ranked alphabet C we define $C^\circ = C \cup \{\circ\}$.

Definition 17 Let $M = (F, Sub, C, \Pi, R_1, R_2, r_{in})$ be an n-sntt. The n-sntt $M^\circ = (F, Sub, C^\circ, \Pi, R_1^\circ, R_2^\circ, r_{in})$ is defined by

– if R_1 contains a rule $f\ (c\, x_1 \ldots x_k) = rhs_{R_1,f,c}$, then R_1° contains a rule $f\ (c\, x_1 \ldots x_k) = \circ\, rhs_{R_1,f,c}$, and
– R_1° contains for every $f \in F$ a (dummy; never applied) rule $f\ (\circ\, x_1) = \ldots$,
– R_2° contains the rules of R_2, and
– R_2° contains the rule $sub\ (\circ\, x_1)\ y_1 \ldots y_n = \circ\ (sub\ x_1\ y_1 \ldots y_n)$.

Let $M{=}(F, C, R, r_{in})$ be an n-satt. The n-satt $M^\circ{=}(F, C^\circ, R^\circ, r_{in})$ is defined by

– if R contains a rule $f\ (c\, x_1 \ldots x_k)\ y_1 \ldots y_n = rhs_{R,f,c}$, then R° contains a rule $f\ (c\, x_1 \ldots x_k)\ y_1 \ldots y_n = \circ\, rhs_{R,f,c}$, and
– R° contains for every $f \in F$ a (dummy) rule $f\ (\circ\, x_1)\ y_1 \ldots y_n = \ldots$. □

Note that by the additional *sub*-rule of R_2° in the previous definition the $\circ$-symbols produced by R_1°-rules are retained.

Definition 18 Let C be a ranked alphabet.
The function $\underline{step} : E_{C^\circ,n}(Z_n) \to I\!N$ is defined as follows:

– For every $e \in E_{C^\circ,n}(Z_n)$: $\underline{step}(\circ\, e) = 1 + \underline{step}(e)$.
– For every $c \in C^{(k)}$ and $e_1, \ldots, e_k \in E_{C^\circ,n}(Z_n)$: $\underline{step}(c\, e_1 \ldots e_k) = \sum_{i=1}^{k} \underline{step}(e_i)$.
– For every $e_1, \ldots, e_n, e \in E_{C^\circ,n}(Z_n)$: $\underline{step}(e[z_i \rightsquigarrow e_i]) = \underline{step}(e) + \sum_{i=1}^{n}(\underline{step}(e_i) * \underline{rel}(z_i, e))$.
– For every $i \in [n]$: $\underline{step}(z_i) = 0$.

The function $\underline{rel} : Z_n \times E_{C^\circ,n}(Z_n) \to \{0,1\}$ is for every $i \in [n]$ and $e \in E_{C^\circ,n}(Z_n)$ defined by $\underline{rel}(z_i, e) = 1$ iff z_i occurs in $\underline{tree}(e)$. □

Example 19 Since the phenomenon of non-relevant subexpressions does not occur in our running example, we choose an artificial example here:

$$\begin{aligned} & \underline{step}((\circ\, z_1)[z_1 \rightsquigarrow (\circ\, A)][z_1 \rightsquigarrow (\circ\, A)]) \\ = {} & \underline{step}((\circ\, z_1)[z_1 \rightsquigarrow (\circ\, A)]) + \underline{step}(\circ\, A) * \underline{rel}(z_1, (\circ\, z_1)[z_1 \rightsquigarrow (\circ\, A)]) \\ = {} & \underline{step}(\circ\, z_1) + \underline{step}(\circ\, A) * \underline{rel}(z_1, (\circ\, z_1)) + 1 * 0 = 1 + 1 * 1 + 1 * 0 = 2 \end{aligned}$$ □

Now we have to consider again our three transformation steps, where the second step involves a formal proof and the first and last step are argumentations concerning call-by-need reduction, which we avoided to define formally.

5.1 Preprocessing

For an n-ntt $M = (F, Sub, C, \Pi, R_1, R_2, r_{in})$ and a term $t \in T_C$ we will denote by $cbn_{R_1}(t)$ the number of R_1-reduction steps which are used to reduce $r_{in}[x_1/t]$ to a term graph corresponding to $nf(\Rightarrow_{R_1 \cup R_2}, r_{in}[x_1/t])$ with a call-by-need reduction. Let $\tilde{M} = (F, \{sub\}, C, \Pi, \tilde{R}_1, \tilde{R}_2, \tilde{r}_{in})$ result from M by preprocessing. Then we have to argue that

$$cbn_{R_1}(t) = \underline{step}(nf(\Rightarrow_{\tilde{R}_1^\circ \cup \tilde{R}_2^\circ}, \tilde{r}_{in}[x_1/t])).$$

First we only consider the F-functions: Since every application of a rule in $\tilde{R}_1^\circ$ delivers exactly one occurrence of $\circ$, the number of occurrences of $\circ$ in $nf(\Rightarrow_{\tilde{R}_1^\circ}, \tilde{r}_{in}[x_1/t])$ equals the number of applied $\tilde{R}_1$-steps to calculate $nf(\Rightarrow_{\tilde{R}_1}, \tilde{r}_{in}[x_1/t])$. This number is in turn equal to the number of applied R_1-steps to calculate a term graph corresponding to $nf(\Rightarrow_{R_1}, r_{in}[x_1/t])$ with call-by-need, because occurrences of F-functions are not nested and hence are neither copied nor deleted.

Now we additionally consider the substitution function sub: (i) Occurrences of F-functions in the recursion argument of sub are neither copied nor deleted in a reduction by $R_1 \cup R_2$. Correspondingly, in a reduction by $\tilde{R}_1^\circ \cup \tilde{R}_2^\circ$ occurrences of $\circ$ in the recursion argument of sub are exactly once reproduced and counted by $\underline{step}$. (ii) In a call-by-need reduction by $R_1 \cup R_2$ no application of an F-function inside a context argument of sub is copied and also in a reduction by $\tilde{R}_1^\circ \cup \tilde{R}_2^\circ$ (with explicit sharing) no corresponding occurrence of $\circ$ is copied. (iii) But, in a call-by-need reduction by $R_1 \cup R_2$, every R_1-step which constitutes a subgraph of the term graph corresponding to $nf(\Rightarrow_{R_1}, r_{in}[x_1/t])$, such that the subgraph occurs in a deleted context argument position j of an occurrence of sub will not be executed, whereas a reduction by $\tilde{R}_1^\circ \cup \tilde{R}_2^\circ$ may behave differently: either the occurrence of $\circ$ in $nf(\Rightarrow_{\tilde{R}_1^\circ}, \tilde{r}_{in}[x_1/t])$ that corresponds to the above R_1-step is also deleted (by a sub-rule on a Π_i with $i \neq j$) or it is shifted into a subexpression e_j of an expression of the form $e[z_i \rightsquigarrow e_i]$ in which z_j does not occur freely in e and hence it is not counted by $\underline{step}$.

5.2 Main Transformation

The proof[7] of efficiency non-deterioration of the main transformation is based on the following lemma. Note the similarity of this lemma to Lemma 13: Instead of the reduction relations of M and $acc(M)$, their "$\circ$-generating versions" are used here. Moreover, instead of calculating the output tree by $\underline{tree}$, the number of $\circ$-symbols is counted by $\underline{step}$.

Lemma 20 Let $M = (F, Sub, C, \Pi, R_1, R_2, r_{in})$ be an n-sntt and $acc(M) = (acc(F), C, acc(R_1), acc(r_{in}))$. For every $f \in F$ and $t \in T_C$:

$$\underline{step}(nf(\Rightarrow_{R_1^\circ \cup R_2^\circ}, sub\ (f\ t)\ z_1 \ldots z_n)) = \underline{step}(nf(\Rightarrow_{acc(R_1)^\circ}, f\ t\ z_1 \ldots z_n)). \quad \square$$

Theorem 21 Let $M = (F, Sub, C, \Pi, R_1, R_2, r_{in})$ be an n-sntt and $acc(M) = (acc(F), C, acc(R_1), acc(r_{in}))$. Then, for every $t \in T_C$:

$$\underline{step}(nf(\Rightarrow_{R_1^\circ \cup R_2^\circ}, r_{in}[x_1/t])) = \underline{step}(nf(\Rightarrow_{acc(R_1)^\circ}, acc(r_{in})[x_1/t])). \quad \square$$

5.3 Postprocessing

Let-expressions do not cause additional reduction steps, rather they denote in functional languages explicit sharings. Thus, if we denote for an n-satt $M = (F, C, R, r_{in})$, for $\tilde{R}$ and $\tilde{r}_{in}$ obtained from R and r_{in}, respectively, by introducing let-expressions, and for a term $t \in T_C$, by $cbn_{\tilde{R}}(t)$ the number of call-by-need

[7] Available at `www.orchid.inf.tu-dresden.de/gdp/conferences/amast06.shtml`

reduction steps which are used to reduce $\tilde{r}_{in}[x_1/t]$ to a term graph corresponding to $nf(\Rightarrow_R, r_{in}[x_1/t])$ with $\tilde{R}$, then it suffices to argue that

$$cbn_{\tilde{R}}(t) = \underline{step}(nf(\Rightarrow_{R^\circ}, r_{in}[x_1/t])).$$

The introduction of let-expressions does not change the copying or deletion properties of the rules in R. Moreover, in a call-by-need reduction by $\tilde{R}$ no function application inside some context argument is copied, and because of the context-linearity of R and hence also of R°, no corresponding occurrence of $\circ$ is copied in a reduction by R°. Hence there is only one main difference in the calculations of $\tilde{R}$ and R°: On the one hand, in a call-by-need reduction by $\tilde{R}$ a function application is not evaluated, if it occurs in a subexpression e_j of an expression of the form $let\ \{v_1 = e_1; \ldots; v_n = e_n\}\ in\ e$, in which e is evaluated and v_j does not occur in e. On the other hand, the corresponding function application in a subexpression e'_j of an expression of the form $e'[z_i \rightsquigarrow e'_i]$ (where $v_1, \ldots, v_n$ correspond to $z_1, \ldots, z_n$, respectively) is evaluated by R° and a symbol $\circ$ is produced. But since z_j does not occur freely in e', the produced $\circ$ is not counted by $\underline{step}$.

6 Future Work

We have proved that our accumulation technique does not deteriorate the efficiency, where efficiency is measured in the number of performed call-by-need reduction steps. This point of view neglects the actual complexity of reduction steps. In particular, we weigh applications of unary functions against applications of the corresponding functions with accumulating parameters. A more elaborate efficiency measure could be based on weighted reduction steps, e.g., by using more than one $\circ$-symbol for a rule with parameters. Furthermore, it would be interesting to develop a syntactic characterization of those programs, for which the time-complexity is changed by accumulation (like in our running example).

In [22] list manipulating operations, in particular append, are eliminated by employing shortcut deforestation [13,7] instead of tree transducer composition as in [16]. To this end, the technique from [22] does not only abstract from list constructors, but also from the list manipulating operations. We believe that this transformation can be generalized to eliminate also tree manipulating operations as, e.g., substitutions. But, as already stated in the Conclusion of [22], “a general statement about the relation between the runtimes of original and transformed programs is hard to make”. Nevertheless it would be interesting to compare such a transformation with our approach.

Acknowledgment

The authors would like to thank Janis Voigtländer for suggesting an improved linear notation for term graphs and for carefully reading a draft of this paper.

References

1. M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Levy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

2. Z.M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. A call-by-need lambda calculus. In *POPL'95, Proceedings*, pages 233–246. ACM Press, 1995.
3. F.L. Bauer and H. Wössner. *Algorithmic Language and Program Development.* Springer–Verlag, 1982.
4. R.S. Bird. The promotion and accumulation strategies in transformational programming. *ACM TOPLAS*, 6(4):487–504, 1984.
5. E.A. Boiten. The many disguises of accumulation. Tech. Report 91-26, Dept. of Informatics, University of Nijmegen, 1991.
6. R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *J. Assoc. Comput. Mach.*, 24:44–67, 1977.
7. O. Chitil. Type inference builds a short cut to deforestation. In *ICFP'99, Paris, France, Proceedings*, volume 34, pages 249–260. ACM Sigplan Notices, 1999.
8. 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.
9. J. Engelfriet and H. Vogler. Macro tree transducers. *J. Comput. Syst. Sci.*, 31:71–145, 1985.
10. J. Engelfriet and H. Vogler. Modular tree transducers. *Theoret. Comput. Sci.*, 78:267–304, 1991.
11. J. Engelfriet and H. Vogler. The translation power of top-down tree-to-graph transducers. *J. Comput. Syst. Sci.*, 49:258–305, 1994.
12. J. Giesl, A. Kühnemann, and J. Voigtländer. Deaccumulation — Improving provability. In *ASIAN'03, Mumbai, India, Proceedings*, volume 2896 of *LNCS*, pages 146–160. Springer-Verlag, 2003.
13. 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.
14. J. Hughes. A novel representation of lists and its application to the function "reverse". *Information Processing Letters*, 22:141–144, 1986.
15. 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, 1999.
16. A. Kühnemann, R. Glück, and K. Kakehi. Relating accumulative and non-accumulative functional programs. In *RTA'01, Utrecht, The Netherlands, Proceedings*, volume 2051 of *LNCS*, pages 154–168. Springer-Verlag, 2001.
17. A. Maletti. Direct construction and efficiency analysis for the accumulation technique for 2-modular tree transducers. Master's thesis, TU Dresden, 2002.
18. H. Partsch. *Specification and Transformation of Programs – A Formal Approach to Software Development.* Springer–Verlag, 1990.
19. W.C. Rounds. Mappings and grammars on trees. *Math. Syst.Th.*, 4:257–287, 1970.
20. D. Sands. A naïve time analysis and its theory of cost equivalence. *Journal of Logic and Computation*, 5(4):495–541, 1995.
21. J.W. Thatcher. Generalized^2 sequential machine maps. *J. Comput. Syst. Sci.*, 4:339–367, 1970.
22. J. Voigtländer. Concatenate, reverse and map vanish for free. In *ICFP'02, Pittsburgh, USA, Proceedings*, pages 14–25. ACM Press, 2002.
23. J. Voigtländer. Formal efficiency analysis for tree transducer composition. Tech. Report TUD-FI04-08, TU Dresden, 2004. To appear in *Theory Comput. Syst.*
24. J. Voigtländer. *Tree Transducer Composition as Program Transformation.* PhD thesis, TU Dresden, 2004.
25. P. Wadler. The concatenate vanishes. Note, University of Glasgow, 1987 (Revised, 1989).