Using Circular Programs to Deforest in Accumulating Parameters

Janis Voigtländer*
Department of Computer Science
Dresden University of Technology
01062 Dresden, Germany
voigt@tcs.inf.tu-dresden.de

ABSTRACT

Functional languages allow a modular programming style by function composition, which however can lead to inefficient runtime behavior due to production and consumption of intermediate results. We present a new mechanizable transformation technique for removing intermediate data structures in the composition of two functions from a class of recursive functions with accumulating parameters, for which classical deforestation techniques fail.

In order to avoid multiple traversals of the input data structure, the composition algorithm produces circular programs that make essential use of lazy evaluation and local recursion. The resulting programs are simplified using a post-processing phase presented in the paper.

Categories and Subject Descriptors

D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features; D.3.4 [**Programming Languages**]: Processors—*optimization*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*program and recursion schemes*

General Terms

Languages, Algorithms

Keywords

Accumulating parameters, circular programs, deforestation, intermediate data structures, lazy composition, lazy evaluation, multiple traversals, program transformation, tupling

*Research supported by the "Deutsche Forschungsgemeinschaft" under grant KU 1290/2-1 and by the "Gesellschaft von Freunden und Förderern der TU Dresden" with a travel grant.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ASIA-PEPM'02, September 12-14, 2002, Aizu, Japan.
Copyright 2002 ACM 1-58113-458-4/02/0009 ...$5.00
This is the author's version of the work. The definitive version is available from http://doi.acm.org/10.1145/568173.568187. An extended version appeared in *Higher-Order and Symbolic Computation*, 17:129–163, 2004.

1. INTRODUCTION

Lazy functional languages are well suited for a modular programming style, where a task is solved by combining solutions of subproblems. This style simplifies the design and verification of programs, and encourages reuse. Unfortunately, modular programs often lack efficiency compared to other — often less understandable — programs that solve the same tasks. These inefficiencies are caused, e.g., by multiple traversals of input data structures and by the production and consumption of structured intermediate results such as lists or trees.

Bird [1] described how *multiple traversals* from nested function calls can be avoided by manually creating *circular programs*. Several traversals of the same input data structure are fused by tupling their results and applying *unfold/fold*-transformation steps [2]. Possible *intra-traversal dependencies* — if information gathered in one traversal is used in another — are captured by circular definitions in the transformed program, which under certain conditions are well-behaved if a lazy evaluation strategy is used. Johnsson [14], Kuiper & Swierstra [18] advocate *attribute grammars* for better understanding the involved transformation. Multiple-traversal programs are reformulated as attribute grammars, which then are efficiently implemented as circular programs in a lazy functional language. Recently, Chin *et al.* [4] presented a *strictness-guided tupling* algorithm that allows for Bird's transformation to be automated.

On the other hand, we often do not have two traversals of the *same* input data structure, but rather a composition of two functions, the first of which traverses the input to produce an *intermediate data structure* that is then traversed by the second function. As an example, consider the following definitions in **Haskell**:

$$\mathbf{data}\ \mathsf{Tree}\ \alpha\ =\ \mathsf{Tip}\ \alpha \mid \mathsf{Node}\ \alpha\ (\mathsf{Tree}\ \alpha)\ (\mathsf{Tree}\ \alpha)$$

$$\begin{aligned}
&pre\ ::\ \mathsf{Tree}\ \alpha \to [\alpha] \to [\alpha]\\
&pre\ (\mathsf{Tip}\ x)\ \ ys\ = x : ys\\
&pre\ (\mathsf{Node}\ x\ xl\ xr)\ ys\ = x : (pre\ xl\ (pre\ xr\ ys))
\end{aligned}$$

$$\begin{aligned}
&rev\ ::\ [\alpha] \to [\alpha] \to [\alpha]\\
&rev\ []\ \ zs\ = zs\\
&rev\ (x : xs)\ zs\ = rev\ xs\ (x : zs)
\end{aligned}$$

The functions *pre* and *rev* are defined using *accumulating parameters* and compute the prefix traversal of a binary tree and the reversal of a list, respectively. If we evaluate — for some tree t — the expression

$$e = (rev\ (pre\ t\ [])\ [])\,,$$

an intermediate list is created and then consumed by the *rev*-function. A standard technique for eliminating intermediate results is Wadler's *deforestation* [28], an algorithmic instance of the *unfold/fold*-technique. However, classical deforestation does not succeed here, due to its well-known problem of not reaching accumulating parameters [3].

Kühnemann [16], Correnson *et al.* [6] tackled this problem by a program transformation approach based on attribute grammars. Primitive recursive function definitions are transformed into attribute grammars, respectively the more abstract *attributed tree transducers* [8]. Under certain conditions two attribute grammars can be composed into a single one, which can then be transformed back into a functional program that does not use an intermediate result. Kakehi *et al.* [15] handle a restricted instance of attribute grammar composition by using a single local rule — similar to the *foldr/build*-rule of *shortcut fusion* [9] — to eliminate intermediate lists in compositions of *map*-style list transformers.

In [25] we have developed — and proved correctness of — a direct composition construction on the level of functional programs without the above indirection via attribute grammars. This direct construction (see also Voigtländer & Kühnemann [27]) has benefits both for the implementation and the accessibility of the transformation and simplifies formal efficiency analysis [26]. Also, it generalizes the technique of attribute grammar composition by handling less restricted programs.

The considered functional programs are extended schemes of primitive recursion — allowing mutual recursion and nesting of terms in parameter positions — so called *macro tree transducers* [7]. Under appropriate restrictions we can compose two macro tree transducers into a single one. The resulting programs are not always satisfactory, because sometimes the price for eliminating the intermediate data structure is that multiple traversals of the input data structure become necessary. Also, it can happen that redundant function calls are created due to a loss of potential sharing in the construction, which is unavoidable if we want to compose into macro tree transducers (which are, by definition, terminating even under eager evaluation). Simply applying the tupling transformation strategy [20, 1, 21, 4] to the resulting programs does not help in general. In the present paper we circumvent these problems by composing into a circular program, thus integrating the tupling with the composition transformation.

For the sake of simplicity we do not consider mutual recursion here, restricting ourselves to "single-state macro tree transducer"-like functions, referred to as *mtt-functions* in the following. Such an mtt-function is defined by pattern matching on its first argument, where the right-hand side of every defining equation may contain the variables from the equation's left-hand side, data constructors, recursive function calls (the first arguments of which must be variables from the pattern match on the left-hand side) and external calls to other functions. Since many typical functions on algebraic data types are defined by such a structural descent, mtt-functions represent a large class of functional programs using accumulating parameters.

We develop a *lazy composition* algorithm that eliminates all intermediate results — including those built up in accumulating parameters — in the composition of two mtt-functions with appropriate restrictions. For the introductory example, our technique will transform the expression e into

$$e'' = (\mathbf{let}\ (c, c_{1,1}) = \overline{prerev}'\ t\ []\ c_{1,1}\ \mathbf{in}\ c)\,,$$

where the following function definition for $\overline{prerev}'$ is produced:

$$\begin{array}{l}
\overline{prerev}' :: \mathsf{Tree}\,\alpha \to [\alpha] \to [\alpha] \to ([\alpha],[\alpha])\\
\overline{prerev}'\ (\mathsf{Tip}\ x)\quad zs\ ys' = (ys', x : zs)\\
\overline{prerev}'\ (\mathsf{Node}\ x\ xl\ xr)\ zs\ ys' = (v_2, v_5)\\
\qquad \mathbf{where}\ (v_2, v_3) = \overline{prerev}'\ xl\ (x : zs)\ v_4\\
\qquad\qquad\quad\ (v_4, v_5) = \overline{prerev}'\ xr\ v_3\quad ys'.
\end{array}$$

Here the intermediate list has been eliminated and only one traversal of the input tree is necessary. In this example an additional post-processing step, which will only be considered informally in this paper, yields the expression

$$e''' = (\overline{prerev}''\ t\ [])$$

with function definition:

$$\begin{array}{l}
\overline{prerev}'' :: \mathsf{Tree}\,\alpha \to [\alpha] \to [\alpha]\\
\overline{prerev}''\ (\mathsf{Tip}\ x)\quad zs = x : zs\\
\overline{prerev}''\ (\mathsf{Node}\ x\ xl\ xr)\ zs = \overline{prerev}''\ xr\ (\overline{prerev}''\ xl\ (x : zs)).
\end{array}$$

The remainder of this paper is organized as follows. Section 2 describes the functional language and mtt-functions. Section 3 presents the ideas of lazy composition, and Section 4 the details of the algorithm. In Section 5 we show the application of the algorithm to our introductory example, consider post-processing steps and efficiency issues. Section 6 informally compares lazy composition with classical deforestation and shortcut fusion. Section 7 concludes and contains future research topics. An appendix uses further examples to discuss extensions of lazy composition.

2. FUNCTIONAL LANGUAGE

We consider programs in a first-order subset of the lazy functional programming language **Haskell**, typed using the Hindley-Milner polymorphic type system [19]. We disallow partial application, hence each data constructor C and function symbol f has a fixed arity. The considered functions have defining equations as given by the grammar in Figure 1, where n and m range over natural numbers and $v, v_1, \ldots, v_n$ range over variable names. Whenever possible without causing confusion, brackets are omitted.

$$\begin{array}{lll}
\varepsilon & ::= f\ \pi\ v_1 \cdots v_n\ =\ \phi & \text{— equation}\\
 & \qquad\qquad \mathbf{where}\ \beta_1 \cdots \beta_m & \text{— optional bindings,}\\
 & & \quad \text{omitted if } m = 0\\
\pi & ::= (\mathsf{C}\ v_1 \cdots v_n) & \text{— pattern}\\
\phi & ::= v & \text{— variable}\\
 & \ \mid\ (\mathsf{C}\ \phi_1 \cdots \phi_n) & \text{— constructor application}\\
 & \ \mid\ (f\ \phi_1 \cdots \phi_n) & \text{— function call}\\
 & \ \mid\ (\phi_1, \ldots, \phi_n) & \text{— tuple with } n \geq 2\\
\beta & ::= (v_1, \ldots, v_n)\ =\ \phi & \text{— binding with } n \geq 1
\end{array}$$

Figure 1: Functional language.

It is not allowed to have two defining equations for the same function and the same constructor C in the pattern matching. The local bindings in a **where**-clause can be mutually recursive. Additionally, the language has a recursive **let**-construct that will not be used in the considered function definitions, but in the expressions to be evaluated.

The functions on which our lazy composition technique operates must be defined by structural recursion on their first argument, but may use additional arguments to accumulate the result.

Definition 1. Let $\mathcal{G}$ be a set of function symbols. An *mtt-function with external calls to* $\mathcal{G}$ is a function $f \notin \mathcal{G}$ in our language, such that:

- the defining equations of f have no bindings (i.e., $m = 0$ in the rule for ε in Figure 1)
- every function call in the right-hand side of a defining equation of f is either a call to some function $g \in \mathcal{G}$, or a recursive call to f with the first argument being one of the variables from the pattern in the equation's left-hand side.

If $\mathcal{G} = \emptyset$, then f is called an *mtt-function without external calls.* $\diamond$

Note that being an mtt-function is a purely syntactic condition on a function's defining equations, hence mtt-functions are easily identified.

The first argument of an mtt-function is called recursion argument, the others are called context parameters. Correspondingly, for a defining equation with left-hand side

$$f\ (\mathsf{C}\ x_1 \cdots x_k)\ y_1 \cdots y_r\ ,$$

the variables $x_1, \ldots, x_k$ and $y_1, \ldots, y_r$ will be called recursion variables and context variables, respectively. An mtt-function is called *recursion-linear*, if for each of its defining equations the right-hand side contains every *recursion* variable at most once. Analogously for *context-linear*.

Example 1. The functions *pre* and *rev* from the introduction are mtt-functions without external calls. So are the functions defined in Figure 2, except *unp* (abbreviated for *unparse*) which is an mtt-function with external calls to $\mathcal{G} = \{show, (+\!\!+)\}$. The mtt-function *exp* is not recursion-linear. The other given functions are both recursion- and context-linear. $\diamond$

3. IDEAS OF LAZY COMPOSITION

Engelfriet & Vogler [7] considered composition results for macro tree transducers, which for our setting imply that two mtt-functions without external calls can be composed into a single function avoiding an intermediate result, if one of the two original functions has no context parameters. In [25] we have generalized their constructions to the case that both functions use context parameters but fulfill certain linearity restrictions (see also [27]). The lazy composition transformation is based on similar ideas, which we will describe informally in this section.

3.1 Translating Right-Hand Sides of f_1 with f_2

Given a composition of two mtt-functions f_1 and f_2 with r and s context parameters, respectively, we want to construct a new function $\overline{f_1 f_2}$ that computes the same result,

```
data Nat  = Zero | Succ Nat
data Term = Num Int | Var String | Prod Term Term |
            Sum Term Term

(++) :: [α] → [α] → [α]
(++)    []    ys = ys
(++) (x : xs) ys = x : ((++) xs ys)

split :: [α] → [α] → [α] → [α]
split    []    ze zo = ze
split (x : xs) ze zo = split xs (x : zo) ze

exp :: Nat → Nat → Nat
exp   Zero    y = Succ y
exp (Succ x)  y = exp x (exp x y)

unp :: Term → String → String
unp   (Num x)     z = (show x) ++ z
unp   (Var x)     z = x ++ z
unp (Prod x1 x2)  z = unp x1 ('*' : (unp x2 z))
unp (Sum x1 x2)   z = '(' : (unp x1 ('+' : (unp x2 (')' : z))))
```

Figure 2: Example definitions.

but without creating and consuming the intermediate data structure.

In the spirit of the *define/instantiate/unfold/fold*-strategy of Burstall & Darlington [2], we start from the basic idea of *folding* nested calls of the form

$$f_2\ (f_1\ \theta\ \phi_1 \cdots \phi_r)\ \psi_1 \cdots \psi_s$$

to

$$\overline{f_1 f_2}\ \theta\ \phi_1 \cdots \phi_r\ \psi_1 \cdots \psi_s.$$

Reading this replacement backwards as a *definition* for $\overline{f_1 f_2}$, and *instantiating* the recursion argument θ to the pattern $(\mathsf{C}\ x_1 \cdots x_k)$ for every data constructor C (of arity k) for which f_1 is defined, we obtain

$$\begin{aligned}&\overline{f_1 f_2}\ (\mathsf{C}\ x_1 \cdots x_k)\ y_1 \cdots y_r\ z_1 \cdots z_s = \\ &\quad f_2\ \underline{(f_1\ (\mathsf{C}\ x_1 \cdots x_k)\ y_1 \cdots y_r)}\ z_1 \cdots z_s\end{aligned}$$

as candidate equation for $\overline{f_1 f_2}$ at C. Given an equation

$$f_1\ (\mathsf{C}\ x_1 \cdots x_k)\ y_1 \cdots y_r\ =\ \mathsf{rhs}_{f_1,\mathsf{C}}$$

in the program, an *unfold*-step transforms this candidate to:

$$\overline{f_1 f_2}\ (\mathsf{C}\ x_1 \cdots x_k)\ y_1 \cdots y_r\ z_1 \cdots z_s = f_2\ (\mathsf{rhs}_{f_1,\mathsf{C}})\ z_1 \cdots z_s.$$

Its right-hand side can be further manipulated by applying *unfold*-steps using f_2's defining equations and *fold*-steps as introduced above, thus in a sense "translating" $\mathsf{rhs}_{f_1,\mathsf{C}}$ with f_2.

So far, we described a variant of classical deforestation [28] tailored to mtt-functions by implicitly abstracting context parameters with **let**-expressions [17].

Example 2. Consider the application of the strategy discussed so far to the introductory example. We calculate the following two defining equations for $\overline{prerev}$:

$$\begin{aligned}&\overline{prerev}\ (\mathsf{Tip}\ x)\ ys\ zs \\ &= rev\ (\mathsf{rhs}_{pre,\mathsf{Tip}})\ zs \\ &= rev\ (x : ys)\ zs \\ &= rev\ ys\ (x : zs) \qquad \text{— by } \textit{unfolding}\end{aligned}$$

$$
\begin{aligned}
& \overline{prerev}\ (\mathsf{Node}\ x\ xl\ xr)\ ys\ zs \\
=\ & rev\ (\mathsf{rhs}_{pre,\mathsf{Node}})\ zs \\
=\ & rev\ (x : (pre\ xl\ (pre\ xr\ ys)))\ zs \\
=\ & rev\ (pre\ xl\ (pre\ xr\ ys))\ (x : zs) && \text{— by } \textit{unfolding} \\
=\ & \overline{prerev}\ xl\ (pre\ xr\ ys)\ (x : zs) && \text{— by } \textit{folding} \quad \diamond
\end{aligned}
$$

Note that in the previous example only parts of the intermediate result have been removed, namely those that occurred outside recursive calls of *pre*. The problem is that in the last *fold*-step above, the context parameter ($pre\ xr\ ys$) of the outer *pre*-call is simply copied without manipulation. Hence, this intermediate result "inside" the context parameter of the outer call is still produced and will eventually be consumed by a *rev*-call in the base case of $\overline{prerev}$ at Tip (compare Example 7 in Section 6.1).

The solution to the problem is a different treatment of context parameters of recursive f_1-calls during the translation of $\mathsf{rhs}_{f_1,\mathsf{C}}$ with f_2, which will ensure that all relevant parts of $\mathsf{rhs}_{f_1,\mathsf{C}}$ are reached by the translation process.

3.2 Adding Translated Parameters of f_1 to $\overline{f_1 f_2}$

If f_1 is an mtt-function *without external calls*, then $\mathsf{rhs}_{f_1,\mathsf{C}}$ might contain data constructors, the variables $x_1, \ldots, x_k$ and $y_1, \ldots, y_r$, and applications of f_1 to $r+1$ arguments, the first of which is some x_i. Hence, during the translation of $\mathsf{rhs}_{f_1,\mathsf{C}}$ with f_2 we can encounter the following kinds of calls to f_2:

(a) $f_2\ (\mathsf{D}\ \phi_1 \cdots \phi_n)\ \psi_1 \cdots \psi_s$, for a data constructor D

(b) $f_2\ x_i\ \psi_1 \cdots \psi_s$, for a recursion variable x_i

(c) $f_2\ y_h\ \psi_1 \cdots \psi_s$, for a context variable y_h

(d) $f_2\ (f_1\ x_i\ \phi_1 \cdots \phi_r)\ \psi_1 \cdots \psi_s$, for a recursion variable x_i.

In case **(a)** we can simply apply the defining equation given for function f_2 at D, and it will be exactly this *unfold*-step that leads to the elimination of intermediate data structures.

In case **(b)** we can keep the call to f_2 on x_i unchanged, because there is no intermediate result to be eliminated.

In case **(c)** we will *not* preserve the call to f_2 on y_h, because then we would miss the elimination of the intermediate data structure inside the h-th context parameter of f_1. Instead, we go beyond the *unfold*/*fold*-strategy by assuming that — beside $y_1, \ldots, y_r$ and $z_1, \ldots, z_s$ — the function $\overline{f_1 f_2}$ has additional arguments, namely translations of f_1's context parameters with f_2. Hence, the left-hand side of the equation under consideration is extended to

$$\overline{f_1 f_2}\ (\mathsf{C}\ x_1 \cdots x_k)\ y_1 \cdots y_r\ z_1 \cdots z_s\ y_1' \ \cdots\ y_r'$$

and in case **(c)** we will simply output y_h'.

Actually a problem arises here, because we have yet to determine what *are* the correct translations of context parameters of f_1 to be passed to calls of $\overline{f_1 f_2}$.

Example 3. The problem is illustrated by adapting Example 2 to use our proposed extended strategy. We start to calculate the two defining equations for $\overline{prerev}$:

$$
\begin{aligned}
& \overline{prerev}\ (\mathsf{Tip}\ x)\ ys\ zs\ ys' \\
=\ & rev\ (\mathsf{rhs}_{pre,\mathsf{Tip}})\ zs \\
=\ & rev\ (x : ys)\ zs \\
=\ & rev\ ys\ (x : zs) && \text{— by } \textbf{(a)} \\
=\ & ys' && \text{— by } \textbf{(c)}
\end{aligned}
$$

$$
\begin{aligned}
& \overline{prerev}\ (\mathsf{Node}\ x\ xl\ xr)\ ys\ zs\ ys' \\
=\ & rev\ (\mathsf{rhs}_{pre,\mathsf{Node}})\ zs \\
=\ & rev\ (x : (pre\ xl\ (pre\ xr\ ys)))\ zs \\
=\ & rev\ (pre\ xl\ (pre\ xr\ ys))\ (x : zs) && \text{— by } \textbf{(a)} \quad \diamond
\end{aligned}
$$

Now, in order to continue the calculation in the previous example, we would like to create a $\overline{prerev}$-call on xl that should take the *rev*-translation of *pre*'s context parameter ($pre\ xr\ ys$) as additional argument. But in order to perform this translation, we would have to specify the appropriate context parameter for *rev* on ($pre\ xr\ ys$). The general form of this problem is discussed in the next subsection.

3.3 Reaching Accumulating Parameters

Let us consider the case **(d)** from the previous subsection, when we have encountered a nested call

$$f_2\ (f_1\ x_i\ \phi_1 \cdots \phi_r)\ \psi_1 \cdots \psi_s$$

during the translation of $\mathsf{rhs}_{f_1,\mathsf{C}}$ with f_2.

By the idea of *folding*, this should be replaced by a call to function $\overline{f_1 f_2}$ on x_i with the context parameters of f_1 and f_2, and — according to our proposed extension — with the f_2-translations of the context parameters of f_1:

$$\overline{f_1 f_2}\ x_i\ \phi_1 \cdots \phi_r\ \psi_1 \cdots \psi_s\ (f_2\ \phi_1\ ?\ \cdots ?)\ \cdots\ (f_2\ \phi_r\ ?\ \cdots ?).$$

Here, the transformation "reaches" intermediate data structures in f_1's accumulating/context parameters, as opposed to classical deforestation. However, before we go on to translate the $\phi_1, \ldots, \phi_r$ with f_2, we have to fill the question mark slots in the previous expression.

Now, what are we supposed to provide in the places of the question marks? It should be the context parameter values, with which f_2 is expected to "arrive" at occurrences of $\phi_1, \ldots, \phi_r$ during computation of the nested call

$$f_2\ (f_1\ x_i\ \phi_1 \cdots \phi_r)\ \psi_1 \cdots \psi_s$$

encountered above (and thus will clearly depend on the value substituted for x_i). However, it is far from obvious whether we can always provide this information. So, we better first answer the following question in general:

> **?** "Given some recursion argument θ for f_1 and a context variable y_h, can we *uniquely* determine what will be the values in context parameter positions of calls of the form ($f_2\ y_h\ \cdots$) occurring during the computation of ($f_2\ (f_1\ \theta\ y_1 \cdots y_r)\ z_1 \cdots z_s$) ?"

3.4 Conditions for Uniqueness

We claim that the above question can be answered positively, if f_1 is context-linear and f_2 is recursion-linear — or, trivially, if one of the two functions has no context parameters at all — by reasoning as follows:

If f_1 is linear in its context parameters, then for every recursion argument θ at most one occurrence of each of the $y_1, \ldots, y_r$ will appear in the result of ($f_1\ \theta\ y_1 \cdots y_r$), because no context parameters of f_1 are copied. If, moreover, f_2 is recursion-linear, then for every occurrence in this intermediate result — in particular for the unique, if existing, occurrence of a y_h — there is at most one possible manner how it can be reached by a call of f_2 during computation of ($f_2\ (f_1\ \theta\ y_1 \cdots y_r)\ z_1 \cdots z_s$).

But uniqueness is not enough, we also need to know how to effectively compute the requested context parameter values.

This could be solved by constructing additional function definitions (cf. [27]), which however can lead to additional traversals of the input data structure. Hence, we instead adjust the function $\overline{f_1 f_2}$ to not only compute the composition of f_1 and f_2, but also the answers to the above question.

3.5 Returning a Tuple

More precisely,

$$\overline{f_1 f_2}\ \theta\ y_1 \cdots y_r\ z_1 \cdots z_s\ y'_1\ \cdots\ y'_r$$

will return a tuple

$$(c, c_{1,1}, \ldots, c_{r,s}),$$

where — assuming that the $y'_1, \ldots, y'_r$ contain the f_2-translations of $y_1, \ldots, y_r$ — the value in c is the result of

$$f_2\ (f_1\ \theta\ y_1 \cdots y_r)\ z_1 \cdots z_s\,,$$

and every $c_{h,l}$-position contains the l-th context parameter value in occurrences of a call to f_2 on y_h during the computation of $(f_2\ (f_1\ \theta\ y_1 \cdots y_r)\ z_1 \cdots z_s)$, if such a call exists.

How can we compute this additional information in the construction of the defining equation for the $\overline{f_1 f_2}$-function at recursion argument $(\mathsf{C}\ x_1 \cdots x_k)$?

Well, in the discussion of case **(c)** in Section 3.2, we encountered the situation that f_2 reaches a context variable y_h, say with a call

$$f_2\ y_h\ \psi_1 \cdots \psi_s.$$

We made use of the provided f_2-translations of f_1's context parameters and replaced this call by y'_h, discarding the values $\psi_1, \ldots, \psi_s$. But these are exactly the sought context parameters of a call to f_2 on y_h, so instead of discarding them we should output $\psi_1, \ldots, \psi_s$ in the $c_{h,1}, \ldots, c_{h,s}$-positions.

Example 4. We illustrate how the additional information can be determined in the construction of the defining equation for $\overline{prerev}$ at Tip by adapting the first calculation in Example 3:

$$\begin{array}{lll}
\overline{prerev}\ (\mathsf{Tip}\ x)\ ys\ zs\ ys' & & \\
= (rev\ (\mathsf{rhs}_{pre,\mathsf{Tip}})\ zs, ?) & & \\
= (rev\ (x : ys)\ zs, ?) & & \\
= (rev\ ys\ (x : zs), ?) & \text{— by} & \textbf{(a)} \\
= (ys', x : zs) & \text{— by} & \textbf{(c)} \quad \diamond
\end{array}$$

Note that no $c_{h,l}$ will be assigned with two different values, because the right-hand side of every equation

$$f_1\ (\mathsf{C}\ x_1 \cdots x_k)\ y_1 \cdots y_r\ =\ \mathsf{rhs}_{f_1,\mathsf{C}}$$

is linear in the $y_1, \ldots, y_r$, and because f_2 is recursion-linear and hence during translation of $\mathsf{rhs}_{f_1,\mathsf{C}}$ with f_2 no occurrence in $\mathsf{rhs}_{f_1,\mathsf{C}}$ will be visited twice with different context parameters. If for some $c_{h,l}$ no value at all is assigned, this means that no call of f_2 on the h-th context parameter of f_1 can occur for the given recursion argument, hence the $c_{h,1}, \ldots, c_{h,s}$-positions will not be needed anyway.

3.6 Creating Circular Bindings

Let us now conclude the discussion of case **(d)** from Section 3.3. There, we had encountered a nested call of the form

$$f_2\ (f_1\ x_i\ \phi_1 \cdots \phi_r)\ \psi_1 \cdots \psi_s$$

and wanted to replace it by a call

$$\overline{f_1 f_2}\ x_i\ \phi_1 \cdots \phi_r\ \psi_1 \cdots \psi_s\ (f_2\ \phi_1\ ?\ \cdots ?)\ \cdots\ (f_2\ \phi_r\ ?\ \cdots ?)\,,$$

still having to fill the question mark positions.

But now, having decided that $\overline{f_1 f_2}$ will beside the composition of f_1 and f_2 also compute the context parameter values with which f_2 reaches the context parameters of f_1, we can resolve the question marks by creating a binding for fresh variables $v, v_{1,1}, \ldots, v_{r,s}$:

$$\begin{array}{rcl}
(v, v_{1,1}, \ldots, v_{r,s}) & = & \overline{f_1 f_2}\ x_i\ \phi_1 \cdots \phi_r\ \psi_1 \cdots \psi_s \\
& & (f_2\ \phi_1\ v_{1,1} \cdots v_{1,s}) \\
& & \cdots \\
& & (f_2\ \phi_r\ v_{r,1} \cdots v_{r,s})\,,
\end{array}$$

where the first element of the tuple gives us the sought composition.

Example 5. Using bindings as suggested above, we can complete the calculation of the defining equation for $\overline{prerev}$ at Node as follows:

$$\begin{array}{lll}
\overline{prerev}\ (\mathsf{Node}\ x\ xl\ xr)\ ys\ zs\ ys' & & \\
= (rev\ (\mathsf{rhs}_{pre,\mathsf{Node}})\ zs, ?) & & \\
= (rev\ (x : (pre\ xl\ (pre\ xr\ ys)))\ zs, ?) & & \\
= (rev\ (pre\ xl\ (pre\ xr\ ys))\ (x : zs), ?) & \text{— by} & \textbf{(a)} \\
= (v, ?) & \text{— by} & \textbf{(d)} \\
\quad \textbf{where}\ (v, v_{1,1})\ =\ \overline{prerev}\ xl\ (pre\ xr\ ys)\ (x : zs) & & \\
\qquad\qquad\qquad\qquad\qquad\qquad (rev\ (pre\ xr\ ys)\ v_{1,1}) & & \\
= (v, ?) & \text{— by} & \textbf{(d)} \\
\quad \textbf{where}\ (v, v_{1,1})\ =\ \overline{prerev}\ xl\ (pre\ xr\ ys)\ (x : zs)\ v' & & \\
\qquad\qquad\ (v', v'_{1,1})\ =\ \overline{prerev}\ xr\ ys\ v_{1,1}\ (rev\ ys\ v'_{1,1}) & & \\
= (v, v'_{1,1}) & \text{— by} & \textbf{(c)} \\
\quad \textbf{where}\ (v, v_{1,1})\ =\ \overline{prerev}\ xl\ \ (pre\ xr\ ys)\ (x : zs)\ v' & & \\
\qquad\qquad\ (v', v'_{1,1})\ =\ \overline{prerev}\ xr\ ys\qquad\quad v_{1,1}\qquad ys' & & \diamond
\end{array}$$

Note that the above binding for $(v, v_{1,1}, \ldots, v_{r,s})$ is circular. In the following, we will argue that under lazy evaluation the program will nevertheless still terminate, because there is no truly circular dependency:

Assume that in the above binding some $v_{h,l}$ depends circularly on itself. We know that the call to $\overline{f_1 f_2}$ will in the $v_{h,l}$-position of its result tuple compute the l-th context parameter of occurrences of a call to f_2 on the h-th context parameter of f_1 during the computation of the nested call $(f_2\ (f_1\ x_i\ \phi_1 \cdots \phi_r)\ \psi_1 \cdots \psi_s)$. For this to circularly depend on $v_{h,l}$, it must in fact depend on the occurrence of

$$f_2\ \phi_h\ v_{h,1} \cdots v_{h,s}$$

— that is, on the y'_h-position of the call to $\overline{f_1 f_2}$ on x_i — because this is the only place where $v_{h,l}$ occurs in the right-hand side of the above binding.

But this would mean — abstracting from the concrete values $\phi_1, \ldots, \phi_r$ and $\psi_1, \ldots, \psi_s$ — that for some recursion argument replaced for x_i, it occurs during the computation of

$$f_2\ (f_1\ x_i\ y_1 \cdots y_r)\ z_1 \cdots z_s$$

a call of f_2 on y_h, the l-th context parameter of which depends on the f_2-translation of y_h. I.e., a nested call of the form

$$(f_2\ y_h\ \cdots\ (\cdots\ (f_2\ y_h\ \cdots)\ \cdots)\ \cdots)$$

would have to occur. Such a nesting would obviously contradict with the fact that — in our setting of context-linear f_1 and recursion-linear f_2 — the context parameters of occurrences of calls to f_2 on y_h are uniquely determined.

4. THE FULL ALGORITHM

Based on the ideas from the previous section we can now describe the lazy composition algorithm formally by syntax-directed transformation rules. We will argue that our algorithm is terminating and correct.

One detail we have disregarded so far is the fact that not all context parameters of an mtt-function need to have the same type. Hence, it does not necessarily make sense to translate *all* the context parameters of f_1 with f_2 as suggested in Section 3.2. In the following transformation this issue is resolved by utilizing the Hindley-Milner type discipline.

TRANSFORMATION 1. *Let f_1 be an mtt-function without external calls and let f_2 be an mtt-function with external calls to $\mathcal{G}$ (where $f_2 \notin \mathcal{G}$), such that either f_1 is context-linear and f_2 is recursion-linear, or one of the two has no context parameters. Let the arities of f_1 and f_2 be $r+1$ and $s+1$, respectively, and assume that some well-typed expression*

$$e = (f_2\ (f_1\ \theta\ \phi_1 \cdots \phi_r)\ \psi_1 \cdots \psi_s)$$

occurs in the program. Without loss of generality — since a reordering of arguments is possible — let $r' \leq r$ be such that of the context parameters $\phi_1, \ldots, \phi_r$ of f_1, exactly the first r' have the same type as the intermediate result $(f_1\ \theta\ \phi_1 \cdots \phi_r)$[1]*. We replace e by the expression*

$$\begin{array}{ll} e' = (\mathbf{let}\ (c, c_{1,1}, \ldots, c_{r',s}) = & \overline{f_1 f_2}\ \theta\ \phi_1 \cdots \phi_r\ \psi_1 \cdots \psi_s \\ & \quad (f_2\ \phi_1\ c_{1,1} \cdots c_{1,s}) \\ & \quad \cdots \\ & \quad (f_2\ \phi_{r'}\ c_{r',1} \cdots c_{r',s}) \\ \quad \mathbf{in}\ c)\,, & \end{array}$$

where the $c, c_{1,1}, \ldots, c_{r',s}$ are fresh variables and the function $\overline{f_1 f_2}$ is defined by pattern-matching on its first argument as follows. For every equation

$$f_1\ (\mathsf{C}\ x_1 \cdots x_k)\ y_1 \cdots y_r \ = \ \mathsf{rhs}_{f_1,\mathsf{C}}$$

in the program, we produce a defining equation for $\overline{f_1 f_2}$ at the same pattern:

$$\begin{array}{l} \overline{f_1 f_2}\ (\mathsf{C}\ x_1 \cdots x_k)\ y_1 \cdots y_r\ z_1 \cdots z_s\ y'_1\ \cdots\ y'_{r'} \ = \\ \quad (\mathcal{L}[\![f_2\ (\mathsf{rhs}_{f_1,\mathsf{C}})\ z_1 \cdots z_s]\!], c_{1,1}, \ldots, c_{r',s}) \\ \quad \mathbf{where}\ \cdots \end{array}$$

with $\mathcal{L}[\![.]\!]$ defined by the rules given in Figure 3. As a side-effect, $\mathcal{L}[\![.]\!]$ produces binding equations that are collected behind the keyword **where***. If for a $c_{h,l}$ from the $c_{1,1}, \ldots, c_{r',s}$ no binding equation was produced, then we add the binding $c_{h,l} = \perp$.*

The rules in Figure 3 cover all cases, except for the possibility that in $\mathcal{L}[\![f_2\ (\mathsf{D}\ \phi_1 \cdots \phi_n)\ \psi_1 \cdots \psi_s]\!]$ there is no defining equation for f_2 at data constructor D in the program. In this case we are at a position in which also the original program would fail — if it would reach that position at all — so we may safely output:

$$\begin{array}{l} \mathcal{L}[\![f_2\ (\mathsf{D}\ \phi_1 \cdots \phi_n)\ \psi_1 \cdots \psi_s]\!] \\ \quad \rightsquigarrow\ \textit{error}\ \text{“Non-exhaustive patterns in function } f_2\text{”} \quad \diamond \end{array}$$

Note that lazy composition does not require f_1 to be recursion-linear. Hence, it can also handle functions like *exp* from Figure 2, which cannot be transformed into an attribute grammar of the form required for the deforestation method by attribute grammar composition [16, 6].

[1] These are then the only context parameters of f_1 that might potentially be reached by calls of f_2 in the intermediate result, due to the Hindley-Milner type system that allows no polymorphic recursion.

1. $\mathcal{L}[\![v]\!] \rightsquigarrow v$
   for some variable v.
2. $\mathcal{L}[\![\mathsf{C}'\ \xi_1 \cdots \xi_n]\!] \rightsquigarrow \mathsf{C}'\ \mathcal{L}[\![\xi_1]\!]\ \cdots\ \mathcal{L}[\![\xi_n]\!]$
   for some data constructor C'.
3. $\mathcal{L}[\![g\ \xi_1 \cdots \xi_n]\!] \rightsquigarrow g\ \mathcal{L}[\![\xi_1]\!]\ \cdots\ \mathcal{L}[\![\xi_n]\!]$
   for some function symbol $g \in \mathcal{G}$.
4. $\mathcal{L}[\![(\xi_1, \ldots, \xi_n)]\!] \rightsquigarrow (\mathcal{L}[\![\xi_1]\!], \ldots, \mathcal{L}[\![\xi_n]\!])$
5. $\mathcal{L}[\![f\ x_i\ \xi_1 \cdots \xi_n]\!] \rightsquigarrow f\ x_i\ \mathcal{L}[\![\xi_1]\!]\ \cdots\ \mathcal{L}[\![\xi_n]\!]$
   for some $f \in \{f_1, f_2\}$.
6. $\mathcal{L}[\![f_2\ (\mathsf{D}\ \phi_1 \cdots \phi_n)\ \psi_1 \cdots \psi_s]\!]$
   $\rightsquigarrow \mathcal{L}[\![\psi[x_1, \ldots, x_n, z_1, \ldots, z_s \leftarrow \phi_1, \ldots, \phi_n, v_1, \ldots, v_s]]\!]$
   if the program contains the defining equation
   $$f_2\ (\mathsf{D}\ x_1 \cdots x_n)\ z_1 \cdots z_s \ = \ \psi.$$
   Here, the $v_1, \ldots, v_s$ are fresh variables for which binding equations $v_1 = \mathcal{L}[\![\psi_1]\!],\ \ldots,\ v_s = \mathcal{L}[\![\psi_s]\!]$ are produced.
7. $\mathcal{L}[\![f_2\ y_h\ \psi_1 \cdots \psi_s]\!] \rightsquigarrow y'_h$
   and the binding equations $c_{h,1} = \mathcal{L}[\![\psi_1]\!],\ \ldots,\ c_{h,s} = \mathcal{L}[\![\psi_s]\!]$ are produced.
8. $\mathcal{L}[\![f_2\ (f_1\ x_i\ \phi_1 \cdots \phi_r)\ \psi_1 \cdots \psi_s]\!] \rightsquigarrow v$
   and — for fresh variables $v, v_{1,1}, \ldots, v_{r',s}$ — the binding equation
   $$\begin{array}{ll} (v, v_{1,1}, \ldots, v_{r',s}) = & \overline{f_1 f_2}\ x_i\ \phi_1 \cdots \phi_r\ \mathcal{L}[\![\psi_1]\!]\ \cdots\ \mathcal{L}[\![\psi_s]\!] \\ & \quad \mathcal{L}[\![f_2\ \phi_1\ v_{1,1} \cdots v_{1,s}]\!] \\ & \quad \cdots \\ & \quad \mathcal{L}[\![f_2\ \phi_{r'}\ v_{r',1} \cdots v_{r',s}]\!] \end{array}$$
   is produced.

Figure 3: Lazy Composition transformation rules.

4.1 Termination and Correctness

LEMMA 1. *The lazy composition transformation terminates.*

PROOF. We measure an expression ξ — to which we want to apply $\mathcal{L}[\![.]\!]$ — with the pair of natural numbers (p, q), where p is the maximal size[2] of a recursion argument of a call to f_2 in ξ, and q is the size of ξ. With the well-founded order

$$(p_1, q_1) < (p_2, q_2)$$
$$\text{iff}$$
$$p_1 < p_2 \text{ or } (p_1 = p_2 \text{ and } q_1 < q_2)$$

[2] The *size* of an expression ϕ (see Figure 1) is defined as follows: the size of a variable is one, the size of an application of a data constructor or function symbol to subexpressions is one greater than the sum of the sizes of the subexpressions, and the size of a tuple is one greater than the sum of the sizes of its elements.

it is easy to see that every application of a rule from 1–8 in Figure 3 gives rise only to applications of $\mathcal{L}[\![.]\!]$ to expressions with a smaller measure than ξ. Termination of $\mathcal{L}[\![.]\!]$ follows by well-founded induction. $\square$

Here, we do not provide a formal correctness proof for lazy composition. Instead, we will below follow the concepts presented in Section 3 to argue that the original expression e and its replacement e' indeed compute the same value, by justifying each of the transformation rules used to build the defining equations of $\overline{f_1 f_2}$. It would be interesting to investigate whether *improvement theory* [23] can be used to prove total correctness of lazy composition from the argued correctness of local transformation steps.

As motivated in Section 3.1, the defining equation for $\overline{f_1 f_2}$ at some constructor C is obtained by translating the right-hand side of f_1 at C with f_2. The steps performed during this translation are specified formally by the transformation rules 1–8 from Figure 3.

It should be clear that the rules 1–6 are locally equivalence-preserving. The abstraction from the $\psi_1, \ldots, \psi_s$ by introduction of fresh variables $v_1, \ldots, v_s$ and binding equations for them in the unfolding rule 6 — which corresponds to case **(a)** from Section 3.2 — is used to avoid duplicated computations in the resulting program (employing the sharing mechanism of lazy evaluation), in case that ψ is non-linear in the $z_1, \ldots, z_s$.

In rule 7 — corresponding to case **(c)** from the informal discussion — we use the provided translation of y_h with f_2. Also, at this point we know with which context parameters f_2 arrives at occurrences of y_h. Uniqueness of these values — which are exactly the values that we have to provide in the result tuple for the $c_{h,1}, \ldots, c_{h,s}$-positions — is guaranteed by f_1 being context-linear and f_2 being recursion-linear, or one of them having no context parameters, as discussed in Sections 3.4 and 3.5.

In rule 8 we apply the function $\overline{f_1 f_2}$ to recursively compute the composition of f_1 and f_2 on a part of the recursion argument, as discussed in Section 3.3 for case **(d)**. In order to do so, we have to provide the context parameters of f_1 and f_2 and the f_2-translations of those context parameters of f_1 that might be reached by f_2 in the intermediate result $(f_1\ x_i\ \phi_1 \cdots \phi_r)$. For the latter, we need the context parameter values of f_2 on reaching such occurrences of $\phi_1 \cdots \phi_{r'}$. These are also computed by the $\overline{f_1 f_2}$-function, yielding a circular definition that will always be well-behaved under lazy evaluation, as discussed in Section 3.6.

5. PRACTICAL ASPECTS

We apply Transformation 1 to the example from the introduction and consider post-processing and efficiency issues.

Example 6. Consider lazy composition for $f_1 = \mathit{pre}$ and $f_2 = \mathit{rev}$ (with $r = r' = s = 1$). Using Transformation 1 we can replace a call

$$\mathit{rev}\ (\mathit{pre}\ t\ [])\ []$$

in the program by the following expression e':

$$\textbf{let}\ (c, c_{1,1}) = \overline{\mathit{prerev}}\ t\ []\ []\ (\mathit{rev}\ []\ c_{1,1})\ \textbf{in}\ c\,,$$

where the function $\overline{\mathit{prerev}}$ is defined by the equations to be read off from the calculations in Figure 4.

In the resulting program the intermediate list is not used anymore. Although fragments of the intermediate data structure still textually appear in the program — e.g., $(\mathit{pre}\ \ \mathit{xr}\ \ \mathit{ys})$ — they will not be evaluated under lazy evaluation and can thus be removed by *useless variable elimination*. $\diamond$

$$
\begin{array}{lll}
 & \overline{\mathit{prerev}}\ (\mathsf{Tip}\ x)\ \mathit{ys}\ \mathit{zs}\ \mathit{ys}' & \\
= & (\mathcal{L}[\![\mathit{rev}\ (x : \mathit{ys})\ \mathit{zs}]\!], c_{1,1}) & \\
\rightsquigarrow & (\mathcal{L}[\![\mathit{rev}\ \mathit{ys}\ (x : v_1)]\!], c_{1,1}) & \text{— by 6} \\
 & \textbf{where}\ v_1 = \mathcal{L}[\![\mathit{zs}]\!] & \\
\rightsquigarrow^2 & (\mathit{ys}', c_{1,1}) & \text{— by 7,1} \\
 & \textbf{where}\ v_1 = \mathit{zs} & \\
 & \qquad\quad c_{1,1} = \mathcal{L}[\![x : v_1]\!] & \\
\rightsquigarrow^3 & (\mathit{ys}', c_{1,1}) & \text{— by 2,1} \\
 & \textbf{where}\ v_1 = \mathit{zs} & \\
 & \qquad\quad c_{1,1} = x : v_1 & \\
 & & \\
 & \overline{\mathit{prerev}}\ (\mathsf{Node}\ x\ \mathit{xl}\ \mathit{xr})\ \mathit{ys}\ \mathit{zs}\ \mathit{ys}' & \\
= & (\mathcal{L}[\![\mathit{rev}\ (x : (\mathit{pre}\ \mathit{xl}\ (\mathit{pre}\ \mathit{xr}\ \mathit{ys})))\ \mathit{zs}]\!], c_{1,1}) & \\
\rightsquigarrow & (\mathcal{L}[\![\mathit{rev}\ (\mathit{pre}\ \mathit{xl}\ (\mathit{pre}\ \mathit{xr}\ \mathit{ys}))\ (x : v_1)]\!], c_{1,1}) & \text{— by 6} \\
 & \textbf{where}\ v_1 = \mathcal{L}[\![\mathit{zs}]\!] & \\
\rightsquigarrow^2 & (v_2, c_{1,1}) & \text{— by 8,1} \\
 & \textbf{where}\ v_1 = \mathit{zs} & \\
 & \qquad\quad (v_2, v_3) = \overline{\mathit{prerev}}\ \mathit{xl}\ (\mathit{pre}\ \mathit{xr}\ \mathit{ys})\ \mathcal{L}[\![x : v_1]\!] & \\
 & \qquad\qquad\qquad\qquad \mathcal{L}[\![\mathit{rev}\ (\mathit{pre}\ \mathit{xr}\ \mathit{ys})\ v_3]\!] & \\
\rightsquigarrow^4 & (v_2, c_{1,1}) & \text{— by 2,1,8} \\
 & \textbf{where}\ v_1 = \mathit{zs} & \\
 & \qquad\quad (v_2, v_3) = \overline{\mathit{prerev}}\ \mathit{xl}\ \ (\mathit{pre}\ \mathit{xr}\ \mathit{ys})\ (x : v_1)\ v_4 & \\
 & \qquad\quad (v_4, v_5) = \overline{\mathit{prerev}}\ \mathit{xr}\ \mathit{ys}\ \mathcal{L}[\![v_3]\!]\ \mathcal{L}[\![\mathit{rev}\ \mathit{ys}\ v_5]\!] & \\
\rightsquigarrow^3 & (v_2, c_{1,1}) & \text{— by 1,7} \\
 & \textbf{where}\ v_1 = \mathit{zs} & \\
 & \qquad\quad (v_2\ , v_3) = \overline{\mathit{prerev}}\ \mathit{xl}\ \ (\mathit{pre}\ \mathit{xr}\ \mathit{ys})\ (x : v_1)\ v_4 & \\
 & \qquad\quad (v_4\ , v_5) = \overline{\mathit{prerev}}\ \mathit{xr}\ \mathit{ys}\qquad\quad v_3 \qquad \mathit{ys}' & \\
 & \qquad\quad c_{1,1} = v_5 &
\end{array}
$$

Figure 4: Lazy Composition for $f_1 = \mathit{pre}$ and $f_2 = \mathit{rev}$.

5.1 Post-processing

In this subsection we present post-processing steps to get rid of ballast that might be introduced by the lazy composition transformation. We illustrate these steps on the program obtained in Example 6 for the original expression $(\mathit{rev}\ (\mathit{pre}\ t\ [])\ [])$, thus gaining the final program from the introduction.

Firstly, the result of a $\overline{\mathit{prerev}}$-call will never depend on its second argument. To detect this, we can use the sufficient condition that the j-th argument (with $j > 1$) of a function $\overline{f_1 f_2}$ is useless, if in no right-hand side of a defining equation for $\overline{f_1 f_2}$ the j-th argument from the corresponding left-hand side occurs elsewhere than inside the j-th argument positions of recursive calls to $\overline{f_1 f_2}$. Then this useless argument can be removed from every call to $\overline{f_1 f_2}$ and from every defining equation of $\overline{f_1 f_2}$.

Secondly, some of the **where**-bindings in the defining equations of $\overline{\mathit{prerev}}$ can be *inlined*. There are several strategies for inlining [22], in our example we choose to inline v_1 and $c_{1,1}$ in both equations of $\overline{\mathit{prerev}}$.

Applying these two mechanizable post-processing steps to the result of Example 6 gives the program of expression e'' and function $\overline{\mathit{prerev}}'$ from the introduction. Considering this function definition, another optimization becomes possible by realizing that in every terminating call of $\overline{\mathit{prerev}}'$ the first element of the result tuple will be equal to the last argument of the call.

This fact is established by a kind of abstract interpretation similar to the automatic *elimination of copy-states* [25] for macro tree transducers. By replacing in the right-hand sides of defining equations for $\overline{prerev}'$ all constructor terms, recursion variables and calls to other functions than $\overline{prerev}'$ with the special symbol $\star$, we obtain the following version:

$$\begin{array}{lll} \overline{prerev}' & (\mathsf{Tip}\ x) \quad zs\ ys' & = (ys', \star) \\ \overline{prerev}' & (\mathsf{Node}\ x\ xl\ xr)\ zs\ ys' & = (v_2, v_5) \\ & \quad \textbf{where}\ (v_2, v_3) & = \overline{prerev}'\ \star\ \star\ v_4 \\ & \quad\quad\quad\ \ (v_4, v_5) & = \overline{prerev}'\ \star\ v_3\ ys'. \end{array}$$

From the base case $(\mathsf{Tip}\ x)$ we obtain the conjecture that $\overline{prerev}'$ returns in the first element of its result tuple its last argument, while the $\star$-symbol in the second tuple element indicates that some other term than just an argument from the left-hand side is returned there. By applying this conjecture for the recursive calls in the inductive case $(\mathsf{Node}\ x\ xl\ xr)$, the second equation is transformed into:

$$\begin{array}{ll} \overline{prerev}'\ (\mathsf{Node}\ x\ xl\ xr)\ zs\ ys' & = (v_2, v_5) \\ \quad \textbf{where}\ (v_2, v_3) & = (v_4, \star) \\ \quad\quad\quad\ \ (v_4, v_5) & = (ys', \star). \end{array}$$

Inlining the two tupled bindings confirms that also in the inductive case the first element of the result tuple is ys'.

By using the thus gained information about the redundancy of the last argument and first element of the result tuple of $\overline{prerev}'$, we can further optimize e'' to expression e''' with function $\overline{prerev}''$ as given in the introduction. Note that the last optimization might change the termination behavior of programs under lazy evaluation, namely non-terminating programs on infinite data structures might be turned into terminating ones.

5.2 Efficiency Considerations

So far, we have considered the pure prevention of creation and consumption of an intermediate data structure by lazy composition as success. However, the elimination of an intermediate result alone is no guaranty for a better efficiency.

The dependency of the transformed program on lazy evaluation can run counter to benefits that the original program might have gained from strictness analysis. The tupling of function results leads to the building of extra closures, which themselves could be seen as a kind of "intermediate data structures", and can deteriorate the efficiency by requiring additional heap space and execution time.

Often a post-processing phase as described in the previous subsection can eliminate this additional ballast. It would be nice to characterize compositions for which this is the case.

When post-processing is not applicable or in addition to post-processing, there are also possibilities for runtime improvements using a compiler that generates more efficient code for functions returning tupled results. Groningen [10] described and implemented a compiler optimization for the lazy functional language **Clean** that reuses closure and tuple selector nodes in recursive calls of functions that yield multiple results in tuples. He observed massive improvements in execution and garbage collection time for a range of examples. The functions produced by lazy composition fulfill exactly the syntactic conditions that are required in order for this tuple optimization to be applicable. Hence, the mentioned compiler optimization might drastically reduce the price that lazy composition has to pay for eliminating intermediate data structures by introducing tuples. The experimental implementation in the **Clean** compiler does not transform functions using circular dependencies, but according to van Groningen [personal communication, 2002] the application of his tuple optimization to circular programs poses no problems in principle, apart from extra implementation work.

6. RELATED WORK ON ELIMINATING INTERMEDIATE RESULTS

In this section we compare lazy composition with classical deforestation and shortcut fusion. Measurements comparing the efficiency of our running example under different program transformation strategies are summarized in Table 1 (obtained using the profiling facilities of the optimizing Glasgow Haskell Compiler on an UltraSPARC workstation).

6.1 Classical Deforestation

Since mtt-functions may be defined using nesting of terms in context parameter positions and hence are not *treeless* programs, Wadler's original deforestation algorithm [28] does not apply directly to them. This problem can be solved by abstracting context parameters using **let**-expressions explicitly [11] or implicitly [17], which yields an algorithm along the lines of Section 3.1.

In order to illustrate in which sense lazy composition eliminates more intermediate data structures than classical deforestation, we compare the programs produced for the introductory example by classical deforestation and by lazy composition and post-processing, respectively.

Example 7. Consider the deforested program from Example 2. Lazy evaluation for the input $(\mathsf{Node}\ 1\ (\mathsf{Tip}\ 2)\ (\mathsf{Tip}\ 3))$ yields the following computation:

$$\begin{array}{l} \quad \overline{prerev}\ (\mathsf{Node}\ 1\ (\mathsf{Tip}\ 2)\ (\mathsf{Tip}\ 3))\ \underline{[]}\ [] \\ = \overline{prerev}\ (\mathsf{Tip}\ 2)\ (pre\ (\mathsf{Tip}\ 3)\ \underline{[]})\ (1 : []) \\ = rev\ (pre\ (\mathsf{Tip}\ 3)\ \underline{[]})\ (2 : 1 : []) \\ = rev\ (\underline{3 : []})\ (2 : 1 : []) \\ = rev\ \underline{[]}\ (3 : 2 : 1 : []) \\ = 3 : 2 : 1 : []\ , \end{array}$$

where the underlined expressions are parts of the intermediate result that have not been eliminated.

Compare this with the shorter computation for the program produced by lazy composition and post-processing:

$$\begin{array}{l} \quad \overline{prerev}''\ (\mathsf{Node}\ 1\ (\mathsf{Tip}\ 2)\ (\mathsf{Tip}\ 3))\ [] \\ = \overline{prerev}''\ (\mathsf{Tip}\ 3)\ (\overline{prerev}''\ (\mathsf{Tip}\ 2)\ (1 : [])) \\ = 3 : (\overline{prerev}''\ (\mathsf{Tip}\ 2)\ (1 : [])) \\ = 3 : 2 : 1 : [] \quad \diamond \end{array}$$

On the other hand, classical deforestation (using explicit or implicit let-abstractions) is applicable to a wider class of programs than just to mtt-functions like our technique.

6.2 Shortcut Fusion

Shortcut fusion achieves elimination of intermediate results by expressing producers and consumers with certain higher-order, polymorphic combinators, the composition of which can be transformed by *foldr/build-* or *cata/augment-*rules [9, 13].

This means that the consumer of an intermediate result needs to be expressed as a *catamorphism*. Such a represen-

Table 1: Average Execution and Garbage Collection Times for Fully Balanced Trees of Different Heights

	height=10		height=15		height=20	
	Exec.+ GC = Total	+/-	Exec.+ GC = Total	+/-	Exec.+ GC = Total	+/-
rev (*pre t* []) []	0.86 + 0.13 = 0.99ms		29.8 + 55.0 = 84.8ms		968 + 3463 = 4431ms	
deforestation	0.85 + 0.12 = 0.97ms	-2%	29.5 + 53.9 = 83.4ms	-2%	980 + 3447 = 4427ms	
shortcut fusion	0.40 + 0.04 = 0.44ms	-56%	14.1 + 33.0 = 47.1ms	-44%	432 + 2007 = 2439ms	-45%
lazy composition	1.85 + 0.25 = 2.10ms	+112%	40.1 + 53.9 = 94.0ms	+10%	1248 + 2427 = 3675ms	-17%
+ post-processing	0.44 + 0.01 = 0.45ms	-55%	13.8 + 8.1 = 21.9ms	-74%	456 + 596 = 1052ms	-76%

tation can be synthesized from the defining equations of an mtt-function f_2 in a systematic way, where a higher-order catamorphism is used in order to capture context parameters.

Example 8. The list consumer *rev* can be expressed with *foldr* as follows:

$rev\ xs\ zs\ =\ foldr\ (\lambda x\ r\ zs \rightarrow r\ (x : zs))\ id\ xs\ zs \quad \diamond$

On the other hand, all data constructors need to be abstracted uniformly from the producer f_1 of an intermediate result in a polymorphic way. But this is problematic for mtt-functions, because parts of the produced output can be "hidden" in the context parameters. One solution would be to prepare those parts for abstraction via an additional traversal, but subsequent removal of the introduced traversal cannot be guaranteed. The alternative is to use the generalization *augment* of *build*, substituting specific values in place of nullary symbols.

Example 9. The list producer *pre* can be expressed with *augment* as follows:

$$\begin{array}{l} pre\ t\ ys\ =\ augment \\ \quad (\lambda c\ n \rightarrow \textbf{let}\ h\ (\mathsf{Tip}\ x)\ ys = x\ `c`\ ys \\ \qquad\qquad h\ (\mathsf{Node}\ x\ xl\ xr)\ ys = x\ `c`\ (h\ xl\ (h\ xr\ ys)) \\ \qquad \textbf{in}\ \ h\ t\ n)\ ys. \end{array}$$

Now, the *foldr*/*augment*-rule can be applied to the expression $e = rev\ (pre\ t\ [])\ []$ from the introduction. After some inlining, the transformation result looks as follows:

$$\begin{array}{l} (\textbf{let}\ h\ (\mathsf{Tip}\ x)\ ys = \lambda zs \rightarrow ys\ (x : zs) \\ \quad h\ (\mathsf{Node}\ x\ xl\ xr)\ ys = \lambda zs \rightarrow h\ xl\ (h\ xr\ ys)\ (x : zs) \\ \ \textbf{in}\ \ h\ t\ id)\ []. \end{array}$$

Here the intermediate list has in fact been eliminated, but therefor a sequence of suspended function calls was introduced. $\diamond$

Note also that the above strategy fails, e.g., for the context-linear mtt-function *split* from Figure 2, which does not even admit a useful definition in terms of *augment*.

Type-inference based deforestation [5] can build a *worker function* for *split* by abstracting not only from the constructors of the result list, but also from the type of the two context parameters. However, fusion of *split* with a consumer function becomes then only possible in expressions where the constructors from the context parameters of *split* can be abstracted, which is not always the case.

Svenningsson [24] extends shortcut fusion by an interesting *destroy*/*unfoldr*-rule that successfully transforms functions recursing over multiple lists simultaneously, but handles accumulating parameters only for *consumers* of intermediate lists.

Hence, the effect to eliminate intermediate results in accumulating parameters as accomplished by lazy composition cannot be achieved by shortcut fusion in general. On the other hand, there exist functions that are no mtt-functions, but *can* be expressed using the above mentioned polymorphic combinators for the different approaches to shortcut fusion.

7. CONCLUSION AND FUTURE WORK

We have developed a program transformation technique — consisting of lazy composition and post-processing — that is applicable to an easily recognized class of recursive function definitions with accumulating parameters, and eliminates intermediate data structures in the composition of two such functions.

To assess better the utility of lazy composition for practical implementations, sufficient conditions should be studied under which the presented transformation technique guarantees an efficiency improvement. In order to reduce the runtime costs originating from the building of unnecessary closures, it could be useful to augment the lazy composition algorithm with a "strictness-guidance", as performed for the tupling transformation strategy by Chin *et al.* [4].

The composition construction on the level of macro tree transducers [25, 27] allows to handle also mutually recursive functions. We would like to apply the idea of lazy composition to this more general framework, constructing circular programs for the composition of restricted macro tree transducers with more than one state.

The direct composition and post-processing constructions for macro tree transducers have been implemented in the **Haskell**$^+$ program transformation system [12]. We want to include lazy composition and related techniques in the system, with the ultimate goal to integrate these techniques into an optimizing functional compiler.

8. ACKNOWLEDGMENTS

I would like to thank Josef Svenningsson for pointing me to John van Groningen's work and John for discussions regarding the possibilities and limitations of his tuple optimization technique combined with lazy composition.

Thanks are also due to several anonymous reviewers for useful comments and suggestions.

9. REFERENCES

[1] R. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21:239–250, 1984.

[2] R. Burstall and J. Darlington. A transformation system for developing recursive programs. *J. ACM*, 24:44–67, 1977.

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

[4] W. Chin, A. Goh, and S. Khoo. Effective optimisation of multiple traversals in lazy languages. In *Partial Evaluation and Semantics-Based Program Manipulation, San Antonio, Texas, Proceedings*, Technical Report, University of Aarhus, Denmark, pages 119–130, 1999.

[5] O. Chitil. *Type-Inference Based Deforestation of Functional Programs*. PhD thesis, RWTH Aachen, 2000.

[6] L. Correnson, E. Duris, D. Parigot, and G. Roussel. Declarative program transformation: A deforestation case-study. In *Principles and Practice of Declarative Programming, Paris, France, Proceedings*, volume 1702 of *LNCS*, pages 360–377. Springer-Verlag, 1999.

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

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

[9] A. Gill, J. Launchbury, and S. Peyton Jones. A short cut to deforestation. In *Functional Programming Languages and Computer Architecture, Copenhagen, Denmark, Proceedings*, pages 223–232. ACM Press, 1993.

[10] J. Groningen. Optimising recursive functions yielding multiple results in tuples in a lazy functional language. In *Implementation of Functional Languages, Lochem, The Netherlands, Proceedings*, volume 1868 of *LNCS*, pages 59–76. Springer-Verlag, 1999.

[11] G. Hamilton and S. Jones. Extending deforestation for first order functional programs. In *1991 Glasgow Workshop on Functional Programming, Portree, Scotland, Proceedings*, Series of Workshops in Computing, pages 134–145. Springer-Verlag, 1992.

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

[13] P. Johann. Short cut fusion: Proved and improved. In *Semantics, Applications, and Implementation of Program Generation, Florence, Italy, Proceedings*, volume 2196 of *LNCS*, pages 47–71. Springer-Verlag, 2001.

[14] T. Johnsson. Attribute grammars as a functional programming paradigm. In *Functional Programming Languages and Computer Architecture, Portland, Oregon, Proceedings*, volume 274 of *LNCS*, pages 154–173. Springer-Verlag, 1987.

[15] K. Kakehi, R. Glück, and Y. Futamura. On deforesting parameters of accumulating maps. In *Logic Based Program Synthesis and Transformation, Paphos, Cyprus, Proceedings*, volume 2372 of *LNCS*, pages 46–56. Springer-Verlag, 2001.

[16] A. Kühnemann. Benefits of tree transducers for optimizing functional programs. In *Foundations of Software Technology & Theoretical Computer Science, Chennai, India, Proceedings*, volume 1530 of *LNCS*, pages 146–157. Springer-Verlag, 1998.

[17] A. Kühnemann. Comparison of deforestation techniques for functional programs and for tree transducers. In *Functional and Logic Programming, Tsukuba, Japan, Proceedings*, volume 1722 of *LNCS*, pages 114–130. Springer-Verlag, 1999.

[18] M. Kuiper and S. Swierstra. Using attribute grammars to derive efficient functional programs. In *Computing Science in the Netherlands, Amsterdam, The Netherlands, Proceedings*. SION, 1987.

[19] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17:348–375, 1978.

[20] A. Pettorossi. Transformation of programs and use of tupling strategy. In *Informatica, Bled, Yugoslavia, Proceedings*, pages 1–6, 1977.

[21] A. Pettorossi. Derivation of programs which traverse their input data only once. In *Advanced School on Programming Methodologies, Rome, Italy, Proceedings*, pages 165–184. Academic Press, 1987.

[22] S. Peyton Jones and S. Marlow. Secrets of the Glasgow Haskell Compiler inliner. In *Workshop on Implementing Declarative Languages, Paris, France, Proceedings*, 1999.

[23] D. Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Trans. on Prog. Lang. and Systems*, 18:175–234, 1996.

[24] J. Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *International Conference on Functional Programming, Pittsburgh, Pennsylvania, Proceedings*. ACM Press, to appear, 2002.

[25] J. Voigtländer. Composition of restricted macro tree transducers. Master's thesis, Dresden University of Technology, 2001. *http://wwwtcs.inf.tu-dresden.de/∼voigt/mscthesis.ps.gz*.

[26] J. Voigtländer. Conditions for efficiency improvement by tree transducer composition. In *Rewriting Techniques and Applications, Copenhagen, Denmark, Proceedings*, volume 2378 of *LNCS*, pages 222–236. Springer-Verlag, 2002.

[27] J. Voigtländer and A. Kühnemann. Composition of functions with accumulating parameters. Technical Report TUD-FI01-08, Dresden University of Technology, 2001.

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

APPENDIX

A. EXTENDING LAZY COMPOSITION

We use examples to informally sketch some extensions that broaden the applicability of lazy composition.

A.1 Relaxing Linearity Restrictions

Example 10. Assume given a coefficients list of a polynomial, for which we want to create a term (of the algebraic data type Term from Figure 2) according to Horner's rule as, e.g., in:

$$x^3 + a_1x^2 + a_2x + a_3 = a_3 + (a_2 + (a_1 + x)x)x.$$

We can do so with the mtt-function *horn* as defined in Figure 5, where in the first context parameter — initialized with the highest coefficient — we accumulate the result and in the second context parameter we pass the variable name for which the polynomial is defined.

$$
\begin{array}{lllll}
\multicolumn{5}{l}{horn :: [\mathsf{Int}] \to \mathsf{Term} \to \mathsf{String} \to \mathsf{Term}} \\
horn & [] & y_1\ y_2 & = & y_1 \\
horn & (a : as) & y_1\ y_2 & = & horn\ as\ (\mathsf{Sum}\ (\mathsf{Num}\ a) \\
 & & & & \qquad (\mathsf{Prod}\ y_1\ (\mathsf{Var}\ y_2)))\ y_2
\end{array}
$$

$$
\begin{array}{lllll}
\multicolumn{5}{l}{part :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha] \to [\alpha] \to ([\alpha],[\alpha])} \\
part\ p & [] & z_1\ z_2 & = & (z_1, z_2) \\
part\ p & (x : xs) & z_1\ z_2 & = & \mathbf{if}\ (p\ x)\ \mathbf{then}\ (part\ p\ xs\ (x : z_1)\ z_2) \\
 & & & & \qquad\qquad\ \mathbf{else}\ \ (part\ p\ xs\ z_1\ (x : z_2))
\end{array}
$$

$$
\begin{array}{llllll}
\multicolumn{6}{l}{merge :: [\alpha] \to [\alpha] \to [\alpha] \to [\alpha]} \\
merge & [] & [] & ys & = & ys \\
merge & (x_1 : xs_1) & [] & ys & = & ys \\
merge & [] & (x_2 : xs_2) & ys & = & ys \\
merge & (x_1 : xs_1) & (x_2 : xs_2) & ys & = & merge\ xs_1\ xs_2\ (x_1 : x_2 : ys)
\end{array}
$$

Figure 5: More definitions.

In order to output such a term, we want to compose *horn* with the accumulating unparse function *unp* defined in Figure 2. Note that *horn* is not context-linear as required for f_1 in Transformation 1. This does no harm however, because only the second context parameter of *horn* is copied (while $r' = 1$) and — for typing reasons — this y_2 will never be reached by a call of *unp* during the translation of *horn*'s right-hand sides[3].

Hence, Transformation 1 *is* applicable, and after post-processing as in Section 5.1 (without copy-elimination) we obtain for

$$unp\ (horn\ as\ (\mathsf{Num}\ 1)\ \text{“x”})\ \text{“”}$$

the replacement

$$\mathbf{let}\ (c, c_{1,1}) = \overline{hornunp}'\ as\ \text{“x”}\ \text{“”}\ ((show\ 1) +\!\!+\ c_{1,1})\ \mathbf{in}\ c\,,$$

where $\overline{hornunp}'$ has defining equations as in Figure 6. ◇

Similarly, we can weaken the restriction on f_2 in the lazy composition transformation by requiring not linearity in all recursion variables, but only that there are no two different recursive calls to f_2 on the same recursion argument. Then care has to be taken to avoid duplicated computations in the resulting program, e.g., by introducing appropriate additional binding equations in rule 6 of Figure 3.

A.2 External Function Calls from f_1

Example 11. Assume that we want to apply Transformation 1 to $f_1 = unp$ and $f_2 = (+\!\!+)$ from Figure 2. Here f_1 uses external calls. Hence, in addition to cases **(a)**–**(d)** from Section 3.2, the following kind of call can be encountered during the translation of $\mathsf{rhs}_{f_1,\mathsf{C}}$ with f_2:

(e) $f_2\ (g\ \phi_1 \cdots \phi_n)\ \psi_1 \cdots \psi_s$, for an external function g.

This case can be handled, if — by some explicitly given or derived equality — we know how f_2 “distributes over” g. Thus, for our example with $f_2 = (+\!\!+)$, the known equality

$$f_2\ (u +\!\!+ v)\ w = u +\!\!+ (f_2\ v\ w)$$

is used to obtain by lazy composition and post-processing that

$$(unp\ x\ y) +\!\!+ z$$

can be replaced by

$$\overline{unp+\!\!+}''\ x\ (y +\!\!+ z)\,,$$

where $\overline{unp+\!\!+}''$ has the same defining equations as *unp*. ◇

[3]Recall that we only imposed linearity restrictions in order to avoid having two different calls of f_2 on the same context parameter of f_1.

$$
\begin{array}{lllll}
\multicolumn{5}{l}{\overline{hornunp}' :: [\mathsf{Int}] \to \mathsf{String} \to \mathsf{String} \to \mathsf{String}} \\
\multicolumn{5}{l}{\qquad\qquad \to (\mathsf{String}, \mathsf{String})} \\
\overline{hornunp}' & [] & y_2\ z\ y_1' & = & (y_1', z) \\
\overline{hornunp}' & (a : as) & y_2\ z\ y_1' & = & (v_1, {}'{*}' : (y_2 +\!\!+ ({}')' : v_2))) \\
\multicolumn{5}{l}{\quad \mathbf{where}\ (v_1, v_2) = \overline{hornunp}'\ as\ y_2\ z\ ({}'(' : ((show\ a)} \\
\multicolumn{5}{l}{\qquad\qquad\qquad\qquad\qquad\qquad\qquad +\!\!+ ({}'{+}' : y_1')))}
\end{array}
$$

$$
\begin{array}{l}
\overline{prepart}'' :: (\alpha \to \mathsf{Bool}) \to \mathsf{Tree}\ \alpha \to [\alpha] \to [\alpha] \to ([\alpha],[\alpha]) \\
\overline{prepart}''\ p\ \ (\mathsf{Tip}\ x)\ \ z_1\ z_2\ = \\
\quad (\mathbf{if}\ v_1\ \mathbf{then}\ x : z_1\ \mathbf{else}\ z_1, \mathbf{if}\ v_1\ \mathbf{then}\ z_2\ \mathbf{else}\ x : z_2) \\
\quad \mathbf{where}\ v_1 = p\ x \\
\overline{prepart}''\ p\ (\mathsf{Node}\ x\ xl\ xr)\ z_1\ z_2\ = \\
\quad (\mathbf{if}\ v_1\ \mathbf{then}\ v_9\ \mathbf{else}\ v_{12}, \mathbf{if}\ v_1\ \mathbf{then}\ v_{10}\ \mathbf{else}\ v_{13}) \\
\quad \mathbf{where}\ \begin{array}[t]{ll}
v_1 & = p\ x \\
(v_3\ , v_4) & = \overline{prepart}''\ p\ xl\ (x : z_1)\ z_2 \\
(v_6\ , v_7) & = \overline{prepart}''\ p\ xl\ z_1\ (x : z_2) \\
(v_9\ , v_{10}) & = \overline{prepart}''\ p\ xr\ v_3\ v_4 \\
(v_{12}, v_{13}) & = \overline{prepart}''\ p\ xr\ v_6\ v_7
\end{array}
\end{array}
$$

$$
\begin{array}{lllllll}
\multicolumn{7}{l}{\overline{mergesplit}'' :: [\alpha] \to [\alpha] \to [\alpha] \to [\alpha] \to ([\alpha],[\alpha])} \\
\overline{mergesplit}'' & [] & [] & ze\ zo & = & (ze, zo) \\
\overline{mergesplit}'' & (x_1 : xs_1) & [] & ze\ zo & = & (ze, zo) \\
\overline{mergesplit}'' & [] & (x_2 : xs_2) & ze\ zo & = & (ze, zo) \\
\overline{mergesplit}'' & (x_1 : xs_1) & (x_2 : xs_2) & ze\ zo & = & (x_2 : v_2, x_1 : v_3) \\
\multicolumn{7}{l}{\quad \mathbf{where}\ (v_2, v_3) = \overline{mergesplit}''\ xs_1\ xs_2\ ze\ zo}
\end{array}
$$

Figure 6: Results of Extended Lazy Composition.

A.3 Conditional Expressions

In order to make mtt-functions more expressive, we would like to add an **if-then-else**-construct to our functional language. Then we could define, e.g., the function *part* as given in Figure 5. It partitions an input list with respect to some predicate p by accumulating in two context parameters those list elements x for which $(p\ x)$ is True and those for which it is False, respectively.

Note that the first argument of *part* is passed unchanged to every recursive call. Hence, while *part* itself is not a proper mtt-function — even with **if-then-else** being allowed — we can consider $\underline{part\ p}$ as an mtt-function (cf. *higher-order macros* in [28]). A composed function resulting from Transformation 1 will then also be parameterized by p, as is the case for $\overline{prepart}''$ obtained below.

Also note, that even with the weakened definition proposed at the end of Appendix A.1 $\underline{part\ p}$ would not be recursion-linear, because it is twice called on recursion argument *xs* in its second defining equation. This could be allowed however, if the two calls occur in different branches of an **if-then-else**-expression and thus will never be evaluated both. On the other hand, the application of an extended Transformation 1 — which would contain rules for follow-

$$
\begin{array}{lllr}
 & \multicolumn{2}{l}{\overline{merge\,split}\ (x_1 : xs_1)\ (x_2 : xs_2)\ ys\ ze\ zo\ ys'} & \\
= & \multicolumn{2}{l}{(\mathcal{L}[\![split\ (merge\ xs_1\ xs_2\ (x_1 : x_2 : ys))\ ze\ zo]\!], c_{1,1}, c_{1,2})} & \\
\rightsquigarrow & \multicolumn{2}{l}{(v_1, c_{1,1}, c_{1,2})} & \text{— by 8'} \\
 & \textbf{where}\ (v_1, v_2, v_3) & = \overline{merge\,split}\ xs_1\ xs_2\ (x_1 : x_2 : ys)\ \mathcal{L}[\![ze]\!]\ \mathcal{L}[\![zo]\!] & \\
 & & \qquad \mathcal{L}[\![split\ (x_1 : x_2 : ys)\ v_2\ v_3]\!] & \\
\rightsquigarrow^3 & \multicolumn{2}{l}{(v_1, c_{1,1}, c_{1,2})} & \text{— by 1,6} \\
 & \textbf{where}\ (v_1, v_2, v_3) & = \overline{merge\,split}\ xs_1\ xs_2\ (x_1 : x_2 : ys)\ ze\ zo & \\
 & & \qquad \mathcal{L}[\![split\ (x_2 : ys)\ (x_1 : v_5)\ v_4]\!] & \\
 & \qquad v_4 & = \mathcal{L}[\![v_2]\!] & \\
 & \qquad v_5 & = \mathcal{L}[\![v_3]\!] & \\
\rightsquigarrow^3 & \multicolumn{2}{l}{(v_1, c_{1,1}, c_{1,2})} & \text{— by 6,1} \\
 & \textbf{where}\ (v_1, v_2, v_3) & = \overline{merge\,split}\ xs_1\ xs_2\ (x_1 : x_2 : ys)\ ze\ zo & \\
 & & \qquad \mathcal{L}[\![split\ ys\ (x_2 : v_7)\ v_6]\!] & \\
 & \qquad v_4 & = v_2 & \\
 & \qquad v_5 & = v_3 & \\
 & \qquad v_6 & = \mathcal{L}[\![x_1 : v_5]\!] & \\
 & \qquad v_7 & = \mathcal{L}[\![v_4]\!] & \\
\rightsquigarrow^9 & \multicolumn{2}{l}{(v_1, c_{1,1}, c_{1,2})} & \text{— by 7,2,1} \\
 & \textbf{where}\ (v_1, v_2, v_3) & = \overline{merge\,split}\ xs_1\ xs_2\ (x_1 : x_2 : ys)\ ze\ zo\ ys' & \\
 & \qquad v_4 & = v_2 & \\
 & \qquad v_5 & = v_3 & \\
 & \qquad v_6 & = x_1 : v_5 & \\
 & \qquad v_7 & = v_4 & \\
 & \qquad c_{1,1} & = x_2 : v_7 & \\
 & \qquad c_{1,2} & = v_6 &
\end{array}
$$

Figure 7: Lazy Composition for multiple recursive $f_1 = merge$.

ing both branches of a conditional expression — could then in different branches lead to different calls of f_2 on context variables of f_1. Hence, the $\mathcal{L}[\![.]\!]$-function is augmented with a list of conditions, keeping track of the branches that it has followed. This information can then be used in a necessary replacement for rule 7 of Figure 3, namely to create conditional bindings for the $c_{h,l}$-positions of the function result.

Example 12. Assume that for some tree t :: $\mathsf{Tree}\,\alpha$ and some predicate p :: $(\alpha \to \mathsf{Bool})$ we would like to create a pair of lists containing those elements in t that do satisfy p, and those that do not, respectively. Reusing the existing functions *pre* and *part*, this can be done by computing

$$part\ p\ (pre\ t\ [])\ []\ [].$$

Following the above idea for an extended lazy composition transformation — and applying post-processing steps as proposed in Section 5.1 — gives us that we can replace the previous expression by

$$\overline{pre\,part}''\ p\ t\ []\ [] \,,$$

with $\overline{pre\,part}''$ defined as in Figure 6. $\diamond$

A.4 Multiple Recursive f_1

Another interesting extension is to consider functions that recurse over two — or more — arguments simultaneously, such as the well-known *zip*-function or the accumulating *merge*-function in Figure 5.

Example 13. The ideas of lazy composition still apply if *merge* is used as f_1 in Transformation 1. Of course, some rules from Figure 3 have to be adjusted, most notably rule 8 to an 8' as exemplified in Figure 7, where the calculation for one defining equation obtained by lazy composition of $f_1 = merge$ with $f_2 = split$ is given. By calculating the other — simpler — defining equations of $\overline{merge\,split}$ and post-processing analogously to Section 5.1, we obtain that

$$split\ (merge\ xs_1\ xs_2\ [])\ []\ []$$

can be replaced by

$$\textbf{let}\ (c_{1,1}, c_{1,2}) = \overline{merge\,split}''\ xs_1\ xs_2\ []\ []\ \textbf{in}\ split\ []\ c_{1,1}\ c_{1,2}\,,$$

where $\overline{merge\,split}''$ has defining equations as given in Figure 6. $\diamond$