Using Circular Programs to Deforest in Accumulating Parameters *

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

Abstract. This paper presents a functional program transformation that removes intermediate data structures in compositions of two members of a class of recursive functions with accumulating parameters. 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 sketched in the paper. The presented transformation, called *lazy composition*, is compared with related transformation techniques both on a qualitative level and based on runtime measurements. An alternative use of higher-orderedness instead of circularity is also briefly explored.

Keywords: program transformation, intermediate results, accumulating arguments, circular programs, tupling, unfold/fold, (short cut) deforestation, tree transducers

CCS 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*

1. Introduction

Lazy functional languages are well suited for a modular programming style that solves a task by combining solutions of subproblems. This style simplifies the design and verification of programs and encourages reuse. Unfortunately, modular programs are often less efficient than monolithic programs that solve the same tasks. Such inefficiencies are, amongst others, caused by *multiple traversals* of input data structures and by the production and consumption of structured *intermediate results*.

Bird [1] described how multiple traversals originating from nested function calls can be avoided by manually creating *circular programs*.

* This is the author's version of a paper appearing in: *Higher-Order and Symbolic Computation* **17**(1–2), 129–163, © KAP, 2004. The definitive version is available from *http://dx.doi.org/10.1023/B:LISP.0000029450.36668.cb*. A previous version appeared in: *Asian Symposium on Partial Evaluation and Semantics-Based Program Manipulation, Aizu, Japan, Proceedings*, pp. 126–137, © ACM Press, 2002.

† Research supported by the DFG under grants KU 1290/2-1 and KU 1290/2-4.

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 given certain conditions are well-behaved under a lazy evaluation strategy. Johnsson [13], Kuiper & Swierstra [18] advocated *attribute grammars* [15] 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, one often encounters—instead of two traversals of the same input data structure—a composition of two functions, the first of which traverses the input and produces an intermediate data structure that is traversed by the second function, which produces the final result. Consider the following Haskell definitions of an algebraic data type for representing simple arithmetic expressions with constants and addition, and an instance of the Show class for unparsing:

```
data Term = Num Int | Add Term Term

instance Show Term where
  show t = unp t ""
    where unp :: Term → String → String
          unp (Num x)     z = shows[1] x z
          unp (Add x1 x2) z = unp x1 ('+' : unp x2 z)
```

Further, consider a function for transforming an arithmetic term as above into a left-associative sum:

```
asc :: Term → Term → Term
asc (Num x)     y = Add y (Num x)
asc (Add x1 x2) y = asc x1 (asc x2 y)
```

Like *unp*, this function is defined using an *accumulating parameter*. An example evaluation illustrating its use is given in Figure 1.

If one wants to output a thus rearranged term, then the evaluation of an expression

$$e = (unp\ (asc\ t\ (\mathsf{Num}\ 0))\ \text{“”})$$

requires creation (by *asc*) and consumption (by *unp*) of an intermediate result. A standard technique for eliminating intermediate results is

[1] The function *shows* from the standard Haskell prelude is used here to convert an integer value to a string-to-string function, e.g. $shows\ 42\ z = '4' : '2' : z$.

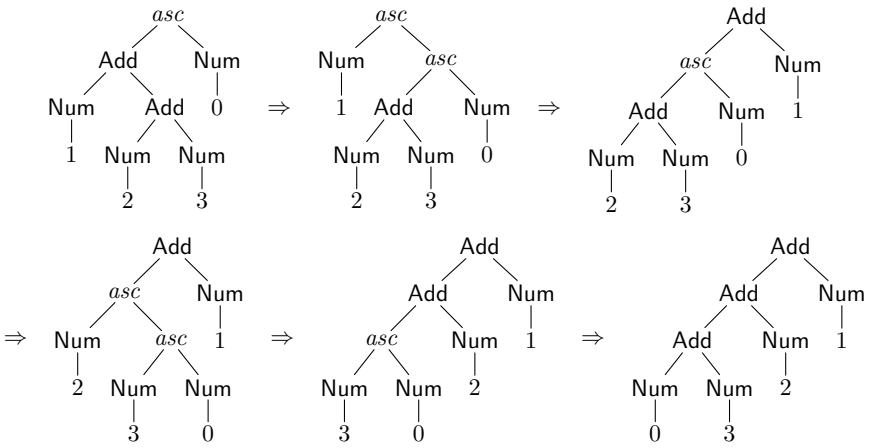

Figure 1. Accumulating parameter of *asc* initialized with (Num 0).

Wadler's *deforestation* [37], 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.* [5] tackled this problem by a program transformation approach based on attribute grammars. Primitive recursive function definitions with accumulating parameters are transformed into attribute grammars, respectively the more abstract *attributed tree transducers* [7]. Under certain conditions, two attribute grammars can be composed into a single one, which can then be transformed back into a functional program that uses no intermediate result. More direct accounts of a related approach on the level of functional programs—without indirection via attribute grammars—have been given by Kakehi *et al.* [14] for a restricted class of *map*-style list transformers, and by Voigtländer & Kühnemann [36] for extended schemes of primitive recursion, so-called *macro tree transducers* [6].

The latter technique of tree transducer composition strictly generalizes attribute grammar composition by handling less restricted programs. The transformation result, however, is not always satisfactory because the price for eliminating the intermediate data structure can be that multiple traversals of the input data structure become necessary. Simply applying the tupling transformation strategy [1, 4, 21, 22] to the resulting programs does not help in general. In the present paper we circumvent this problem 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 the *lazy composition* algorithm, which transforms the composition of two mtt-functions fulfilling appropriate restrictions into a monolithic program. For the running example, our technique transforms the expression e into

$$\overline{e}'' = (\mathbf{let}\ (c, c_{1,1}) = \overline{\mathit{ascunp}}''\ t\ \text{“”}\ (\mathit{shows}\ 0\ c_{1,1})\ \mathbf{in}\ c)\,,$$

additionally producing the following function definition:

$$\begin{array}{l}
\overline{\mathit{ascunp}}''\ ::\ \mathsf{Term} \to \mathsf{String} \to \mathsf{String} \to (\mathsf{String}, \mathsf{String}) \\
\overline{\mathit{ascunp}}''\ (\mathsf{Num}\ x) \quad z\ y' = (y', '+' : \mathit{shows}\ x\ z) \\
\overline{\mathit{ascunp}}''\ (\mathsf{Add}\ x_1\ x_2)\ z\ y' = \mathbf{let}\ (v_1, v_2) = \overline{\mathit{ascunp}}''\ x_1\ z\ \ v_3 \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ (v_3, v_4) = \overline{\mathit{ascunp}}''\ x_2\ v_2\ y' \\
\qquad\qquad\qquad\qquad\qquad\quad\ \ \mathbf{in}\ \ (v_1, v_4)
\end{array}$$

The intermediate result has been eliminated and only one traversal of the input term is necessary. Additional post-processing steps yield the expression

$$\overline{e}'''' = (\mathit{shows}\ 0\ (\overline{\mathit{ascunp}}''''\ t\ \text{“”}))$$

and the following function definition:

$$\begin{array}{l}
\overline{\mathit{ascunp}}''''\ ::\ \mathsf{Term} \to \mathsf{String} \to \mathsf{String} \\
\overline{\mathit{ascunp}}''''\ (\mathsf{Num}\ x) \quad z = '+' : \mathit{shows}\ x\ z \\
\overline{\mathit{ascunp}}''''\ (\mathsf{Add}\ x_1\ x_2)\ z = \overline{\mathit{ascunp}}''''\ x_2\ (\overline{\mathit{ascunp}}''''\ x_1\ z)
\end{array}$$

Measurements confirm that this version is considerably more efficient than the original program consisting of e, *asc*, and *unp*.

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; Section 4, the details of the algorithm. In Section 5 we show the application of the algorithm to our introductory example and consider post-processing steps and variations on the theme of lazy composition. Section 6 informally compares lazy composition with classical deforestation, *shortcut fusion* [9], and tree transducer composition. Section 7 discusses efficiency issues for programs produced by lazy composition and performs a runtime measurements based comparison with the results of other transformation techniques. Section 8 concludes.

2. Functional language

We consider programs in a first-order subset of the lazy functional programming language Haskell [24], typed using the Hindley-Milner polymorphic type system [19]. There is no partial application, so that each data constructor C or function symbol f has a fixed arity. The considered functions are defined by sequences of equations as given by the grammar in Figure 2, 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. It is not allowed for two defining equations of the same function to have the same constructor in their patterns. The local bindings in a **let**-block can be mutually recursive.

$$
\begin{array}{lll}
\delta ::= \varepsilon_1 \cdots \varepsilon_n & & \text{— function definition} \\
\varepsilon ::= f\ \pi\ v_1 \cdots v_n = \textbf{let}\ \beta_1 & & \text{— equation with optional bindings} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \vdots & & \text{\quad (omitted if } m = 0) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \beta_m & & \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ \ \phi & & \\
\pi ::= (\mathsf{C}\ v_1 \cdots v_n) & & \text{— pattern} \\
\phi ::= v & & \text{— variable} \\
\quad \mid\ (\mathsf{C}\ \phi_1 \cdots \phi_n) & & \text{— constructor application} \\
\quad \mid\ (\phi_1, \ldots, \phi_n) & & \text{— tuple with } n \geq 2 \\
\quad \mid\ (f\ \phi_1 \cdots \phi_n) & & \text{— function call} \\
\beta ::= (v_1, \ldots, v_n) = \phi & & \text{— binding with } n \geq 1
\end{array}
$$

Figure 2. Functional language.

The functions that are used as input for our transformation technique must be defined by structural recursion on their first arguments, but may use additional arguments to accumulate their results.

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 local bindings (i.e. $m = 0$ in the rule for ε in Figure 2) and
- every function call in the right-hand side of a defining equation of f is either a call to some $g \in \mathcal{G}$ or a recursive call to f with one of the variables from the pattern in the equation's left-hand side as its first argument.

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 preclusion of local bindings simplifies the description of the restrictions necessary in order to apply our technique and prevents the original program from already being circular.

The first argument of an mtt-function is called its *recursion argument*, the others are called *context parameters*. Correspondingly, in 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 $y_1, \ldots, y_r$ are called recursion variables and context variables, respectively. An mtt-function is called *recursion-linear* if the right-hand side of each of its defining equations contains every *recursion* variable at most once; analogously for *context-linear*.

Example 1. The function *unp* (from the introduction) is an mtt-function with an external call to $\mathcal{G} = \{shows\}$. The function *asc* is an mtt-function without external calls, and so are the functions defined in Figure 3. Except for *exp* (which is context-linear, but not recursion-linear) and *hanoi* (which is neither recursion- nor context-linear), all the given functions are both recursion- and context-linear. $\diamond$

3. Ideas of lazy composition

Engelfriet & Vogler [6] studied macro tree transducers and proved composition results which for our setting imply that two mtt-functions without external calls can be composed into a single mtt-function if one of the two original functions has no context parameters. In [34] we have generalized their constructions to the case that both functions use context parameters, but fulfill certain linearity restrictions (see also [36]). The lazy composition transformation is based on similar ideas, to be described 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, but without producing 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

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

$$\begin{array}{l}
\textbf{data}\ \mathsf{Tree}\ \alpha = \mathsf{Tip}\ \alpha \mid \mathsf{Node}\ \alpha\ (\mathsf{Tree}\ \alpha)\ (\mathsf{Tree}\ \alpha) \\
\textbf{data}\ \mathsf{Nat} \quad = \mathsf{Zero} \mid \mathsf{Succ}\ \mathsf{Nat}
\end{array}$$

$$\begin{array}{l}
pre :: \mathsf{Tree}\ \alpha \to [\alpha] \to [\alpha] \\
pre\ (\mathsf{Tip}\ x) \qquad\quad ys = x : ys \\
pre\ (\mathsf{Node}\ x\ xl\ xr)\ ys = x : (pre\ xl\ (pre\ xr\ ys))
\end{array}$$

$$\begin{array}{l}
rev :: [\alpha] \to [\alpha] \to [\alpha] \\
rev\ [] \qquad zs = zs \\
rev\ (x : xs)\ zs = rev\ xs\ (x : zs)
\end{array}$$

$$\begin{array}{l}
(+\!\!+) :: [\alpha] \to [\alpha] \to [\alpha] \\
(+\!\!+)\ [] \qquad ys = ys \\
(+\!\!+)\ (x : xs)\ ys = x : ((+\!\!+)\ xs\ ys)
\end{array}$$

$$\begin{array}{l}
exp :: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \\
exp\ \mathsf{Zero} \qquad y = \mathsf{Succ}\ y \\
exp\ (\mathsf{Succ}\ x)\ y = exp\ x\ (exp\ x\ y)
\end{array}$$

$$\begin{array}{l}
hanoi :: \mathsf{Int} \to \alpha \to \alpha \to \alpha \to [(\alpha, \alpha)] \to [(\alpha, \alpha)] \\
hanoi\ 0 \qquad\quad y_1\ y_2\ y_3\ ys = ys \\
hanoi\ (x+1)\ y_1\ y_2\ y_3\ ys = hanoi\ x\ y_1\ y_3\ y_2 \\
\qquad\qquad\qquad\qquad\qquad\quad ((y_1, y_2) : (hanoi\ x\ y_3\ y_2\ y_1\ ys))
\end{array}$$

Figure 3. Example definitions.

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[2], we obtain

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

as a 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 original program, an *unfolding* transforms this candidate to:

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

The 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.

[2] Since f_1 and f_2 are strict in their first arguments, so is $\overline{f_1 f_2}$; hence, the instantiation step is safe [28].

So far, we have described a variant of classical deforestation [37], tailored to mtt-functions by implicitly abstracting context parameters with **let**-expressions[3] [17].

Example 2. We apply the strategy discussed so far to the introductory example by calculating the following defining equations for $\overline{ascunp}$:

$$\begin{array}{lll}
& \overline{ascunp}\ (\mathsf{Num}\ x)\ y\ z & \\
= & unp\ (\mathsf{rhs}_{asc,\mathsf{Num}})\ z & \\
= & unp\ (\mathsf{Add}\ y\ (\mathsf{Num}\ x))\ z & \\
\leadsto & unp\ y\ (\mathit{'+'} : unp\ (\mathsf{Num}\ x)\ z) & \text{— by unfolding} \\
\leadsto & unp\ y\ (\mathit{'+'} : shows\ x\ z) & \text{— by unfolding}
\end{array}$$

$$\begin{array}{lll}
& \overline{ascunp}\ (\mathsf{Add}\ x_1\ x_2)\ y\ z & \\
= & unp\ (\mathsf{rhs}_{asc,\mathsf{Add}})\ z & \\
= & unp\ (asc\ x_1\ (asc\ x_2\ y))\ z & \\
\leadsto & \overline{ascunp}\ x_1\ (asc\ x_2\ y)\ z & \text{— by folding.} \quad \diamond
\end{array}$$

Note that in the previous example only parts of the intermediate result are reached by the transformation. Namely, in the recursive case the fold-step—also shown in Figure 4—simply copies the context parameter $(asc\ x_2\ y)$ of the outer call to the producer *asc*, without any manipulation. The transformed program then still requires the explicit construction of this part of the intermediate result and its eventual consumption by an *unp*-call (compare Example 9 in Section 6.1).

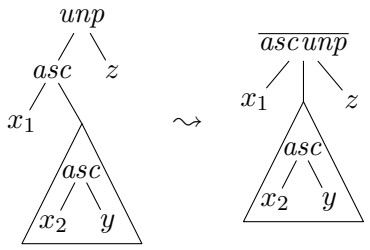

Figure 4. Fold-step of classical deforestation.

We are going to present a solution to this problem that is based on 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] More precisely, a function call of the form $(f\ x_i\ \phi_1 \cdots \phi_n)$ is interpreted as $(\mathbf{let}\ (v_1,\ldots,v_n) = (\phi_1,\ldots,\phi_n)\ \mathbf{in}\ f\ x_i\ v_1 \cdots v_n)$ and the *blazed* deforestation algorithm is applied, transforming **let**-bound expressions separately.

3.2. Adding translated parameters of f_1 to $\overline{f_1 f_2}$

From now on, we assume that f_1 is an mtt-function without external calls. Then, according to Definition 1 and the grammar in Figure 2, $\mathsf{rhs}_{f_1,\mathsf{C}}$ might contain tuples, data constructors, the variables $x_1, \ldots, x_k$ and $y_1, \ldots, y_r$, and recursive calls to f_1 on some x_i. For typing reasons, the first argument of f_2 is never a tuple. Thus, during the translation of $\mathsf{rhs}_{f_1,\mathsf{C}}$ with f_2 we can encounter only 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, and

(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 simply unfold the defining equation given for f_2 at D. Exactly such unfold-steps will lead to the elimination of intermediate data structures. In case **(b)** we can preserve the call to f_2 on x_i because there is no intermediate result to be eliminated here.

In case **(c)**, however, we must not preserve the f_2-call 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-technique by assuming that beside $y_1, \ldots, y_r$ and $z_1, \ldots, z_s$ the function $\overline{f_1 f_2}$ takes additional arguments, namely translations of f_1's context parameters with f_2. By such a translation we mean a complete application of f_2 with appropriate context parameters. Under the assumptions that these could be determined correctly and that the obtained translations are stored in additional variables $y'_1, \ldots, y'_r$, i.e. the left-hand side of the equation under consideration is extended to

$$\overline{f_1 f_2}\ (\mathsf{C}\ x_1 \cdots x_k)\ y_1 \cdots y_r\ z_1 \cdots z_s\ y'_1 \cdots y'_r\,,$$

we can simply output y'_h in case **(c)**.

From this approach the question arises whether it is at all possible to determine "a priori" the correct context parameters of f_2 in translations of context parameters of f_1 to be passed to calls of $\overline{f_1 f_2}$. The problem is illustrated by attempting to adapt Example 2 to use our proposed extended strategy. Its general form will be discussed in the next subsection.

Example 3. The calculation of the defining equation for the base case of $\overline{ascunp}$ is easily adapted:

$$\overline{ascunp}\ (\mathsf{Num}\ x)\ y\ z\ y'$$

$$\begin{array}{ll} = & \mathit{unp}\ (\mathsf{rhs}_{\mathit{asc},\mathsf{Num}})\ z \\ = & \mathit{unp}\ (\mathsf{Add}\ y\ (\mathsf{Num}\ x))\ z \\ \rightsquigarrow & \mathit{unp}\ y\ (\mathit{'+'} : \mathit{unp}\ (\mathsf{Num}\ x)\ z) \quad \text{— by } \mathbf{(a)} \\ \rightsquigarrow & y' \qquad\qquad\qquad\qquad\qquad\quad \text{— by } \mathbf{(c)} \end{array}$$

Note that the last transformation step here is only possible because we assume that y' already stores the *unp*-translation of y with exactly the correct context parameter, in this case $(\mathit{'+'} : \mathit{unp}\ (\mathsf{Num}\ x)\ z)$.

The calculation for the recursive case, however, gets stuck early:

$$\begin{array}{ll} & \overline{\mathit{asc}\,\mathit{unp}}\ (\mathsf{Add}\ x_1\ x_2)\ y\ z\ y' \\ = & \mathit{unp}\ (\mathsf{rhs}_{\mathit{asc},\mathsf{Add}})\ z \\ = & \mathit{unp}\ (\mathit{asc}\ x_1\ (\mathit{asc}\ x_2\ y))\ z \\ \rightsquigarrow & \cdots \qquad\qquad\qquad\qquad\qquad\quad \text{— by } \mathbf{(d)}? \end{array}$$

The $\overline{\mathit{asc}\,\mathit{unp}}$-call on x_1 to be created here should take the translation of *asc*'s context parameter $(\mathit{asc}\ x_2\ y)$ with *unp* as additional argument. To perform this translation, we would first have to specify the appropriate context parameter for *unp* on $(\mathit{asc}\ x_2\ y)$, i.e. to fill the question mark position in Figure 5. ◇

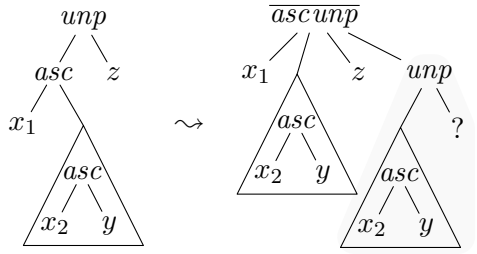

Figure 5. An extended fold-step, missing information about the context parameter of *unp* on the intermediate result produced inside the context parameter of *asc*.

3.3. Reaching accumulating parameters

Let us consider the general 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, taking as further arguments the context parameters of f_1 and f_2, and—according to our proposed extension—the f_2-translations

of the context parameters of f_1:

$$\overline{f_1 f_2}\ x_i\ \phi_1 \cdots \phi_r\ \psi_1 \cdots \psi_s\ (f_2\ \phi_1\ \underbrace{?\ \cdots\ ?}_{s \times \text{“?”}})\ \cdots\ (f_2\ \phi_r\ \underbrace{?\ \cdots\ ?}_{s \times \text{“?”}}).$$

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.

What are we supposed to provide in those places? Clearly, 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 clearly depends on the value substituted for x_i). Since it is not obvious whether we can always provide such information, 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 the values in context parameter positions of calls of the form $(f_2\ y_h\ \cdots)$ occurring during the computation of $(f_2\ (f_1\ \theta\ y_1 \cdots y_r)\ z_1 \cdots z_s)$?

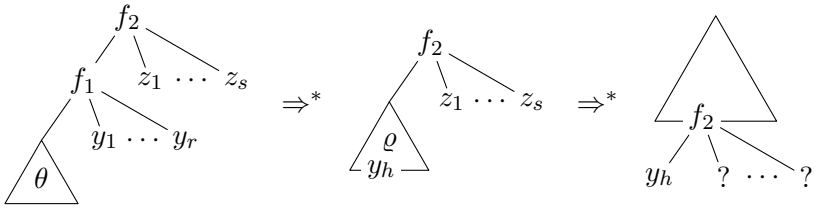

Figure 6. Can we uniquely determine the values in question mark positions?

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 does not copy 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 ϱ of $(f_1\ \theta\ y_1 \cdots y_r)$. If, moreover, f_2 is recursion-linear, then for every position in this intermediate result ϱ there is at most one possible manner how it can be reached by a call of f_2 during computation of $(f_2\ (f_1\ \theta\ y_1 \cdots y_r)\ z_1 \cdots z_s)$.

However, 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. [36]), which can lead to additional traversals of the input data structure though. To avoid this, 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 correct 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 this result, if such a call exists.

To find out how this additional information can be computed in the construction of the defining equation for $\overline{f_1 f_2}$ at recursion argument $(\mathsf{C}\ x_1 \cdots x_k)$, we return attention to the discussion of case **(c)** in Section 3.2. There we considered 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 $\overline{\mathit{ascunp}}$ at Num by further adapting the first calculation from Example 3:

$$\begin{aligned}
& \overline{\mathit{ascunp}}\ (\mathsf{Num}\ x)\ y\ z\ y' \\
= {}& (\mathit{unp}\ (\mathsf{rhs}_{\mathit{asc},\mathsf{Num}})\ z, ?) \\
= {}& (\mathit{unp}\ (\mathsf{Add}\ y\ (\mathsf{Num}\ x))\ z, ?) \\
\rightsquigarrow {}& (\mathit{unp}\ y\ ('+' : \mathit{unp}\ (\mathsf{Num}\ x)\ z), ?) && \text{— by } \mathbf{(a)} \\
\rightsquigarrow {}& (y', '+' : \mathit{unp}\ (\mathsf{Num}\ x)\ z) && \text{— by } \mathbf{(c)}\text{, revised} \\
\rightsquigarrow {}& (y', '+' : \mathit{shows}\ x\ z) && \text{— by } \mathbf{(a)}. \quad \diamond
\end{aligned}$$

Note that in general no $c_{h,l}$ is 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 no position in $\mathsf{rhs}_{f_1,\mathsf{C}}$ can be visited twice with different context parameters during the translation of $\mathsf{rhs}_{f_1,\mathsf{C}}$ with f_2. If for some $c_{h,l}$ no value at all is assigned, then 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, in this case 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 decided to replace it by a call

$$\overline{f_1 f_2}\ x_i\ \phi_1 \cdots \phi_r\ \psi_1 \cdots \psi_s\ (f_2\ \phi_1\ \underbrace{?\ \cdots\ ?}_{s \times \text{“?”}})\ \cdots\ (f_2\ \phi_r\ \underbrace{?\ \cdots\ ?}_{s \times \text{“?”}}),$$

still having to fill the question mark positions.

But now, having decided that $\overline{f_1 f_2}$ returns beside the composition of f_1 and f_2 also 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 $v, v_{1,1}, \ldots, v_{r,s}$:

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

where the first tuple element gives us the sought composition.

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

$$\begin{aligned}
& \overline{ascunp}\ (\mathsf{Add}\ x_1\ x_2)\ y\ z\ y' \\
= {}& (unp\ (\mathsf{rhs}_{asc,\mathsf{Add}})\ z, ?) \\
= {}& (unp\ (asc\ x_1\ (asc\ x_2\ y))\ z, ?) \\
\rightsquigarrow {}& \mathbf{let}\ (v, v_{1,1}) = \overline{ascunp}\ x_1\ (asc\ x_2\ y)\ z \\
& \qquad\qquad\qquad\qquad\quad (unp\ (asc\ x_2\ y)\ v_{1,1}) \\
& \quad \mathbf{in}\ \ (v, ?) \\
\rightsquigarrow {}& \mathbf{let}\ (v, v_{1,1}) = \overline{ascunp}\ x_1\ (asc\ x_2\ y)\ z\ v' \\
& \qquad (v', v'_{1,1}) = \overline{ascunp}\ x_2\ y\ v_{1,1}\ (unp\ y\ v'_{1,1}) \\
& \quad \mathbf{in}\ \ (v, ?) \\
\rightsquigarrow {}& \mathbf{let}\ (v, v_{1,1}) = \overline{ascunp}\ x_1\ (asc\ x_2\ y)\ z\ v' \\
& \qquad (v', v'_{1,1}) = \overline{ascunp}\ x_2\ y\ v_{1,1}\ y' \\
& \quad \mathbf{in}\ \ (v, v'_{1,1}).
\end{aligned}$$

— by **(d)**

— by **(d)**

— by **(c)**

The two performed **(d)**-steps are illustrated in Figure 7, where the elements bound in tuples are represented by explicit occurrences of the pair selectors *fst* and *snd* (with the variable names from the above calculation annotated for clarity).

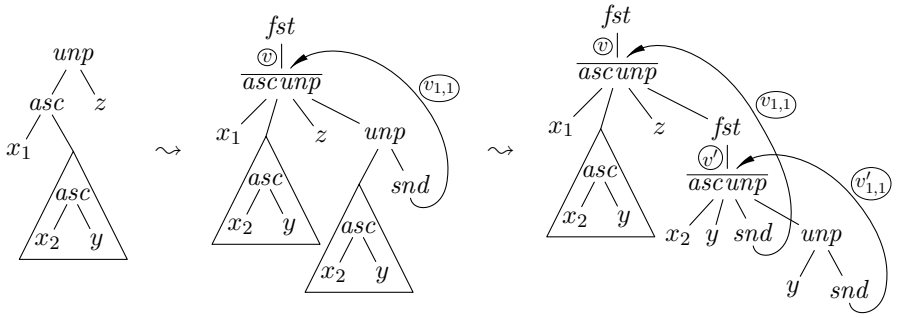

Figure 7. The information missing in Figure 5 determined by using circular bindings.

We have now solved the problem of not reaching intermediate results produced inside context parameters. Note that—beside its translation with *unp*—also the context parameter $(asc\ x_2\ y)$ itself still appears in the transformed program. Such a preservation is necessary for full generality of lazy composition in cases where in the original program parts of the intermediate result are copied literally into the final output by the consuming mtt-function and hence their construction cannot be completely avoided. If this is not the case, then such fragments are removed from the transformed program by a post-processing phase as to be shown for our running example in Section 5.1. $\diamond$

Created bindings such as the one for $(v, v_{1,1}, \ldots, v_{r,s})$ above the previous example are circular, i.e. we have "tied the knot". In the following subsection we argue that nevertheless there are no truly circular data dependencies and hence a lazy evaluation mechanism can order the computations in such a way that the transformed program still terminates.

3.7. Termination under lazy evaluation

To ensure termination of the transformed program, we have to establish that in a created binding of the form

$$(v, v_{1,1}, \ldots, v_{r,s}) = \overline{f_1 f_2}\ x_i\ y_1 \cdots y_r\ z_1 \cdots z_s\ \begin{array}{l}(f_2\ y_1\ v_{1,1} \cdots v_{1,s}) \\ \cdots \\ (f_2\ y_r\ v_{r,1} \cdots v_{r,s})\end{array}$$

no $v_{h,l}$ depends circularly on itself.

Assume the contrary were the case, i.e. such a circular dependency would exist. Then the $v_{h,l}$-position of the result tuple of the above $\overline{f_1 f_2}$-call would have to depend on the call's y'_h-argument[4]. Since the former computes the l-th context parameter of occurrences of a call to f_2 on y_h resulting from $(f_2\ (f_1\ x_i\ y_1 \cdots y_r)\ z_1 \cdots z_s)$—i.e. the l-th question mark position in Figure 6—this would mean that 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 occurs, the l-th context parameter of which depends on the f_2-translation of y_h. Such a situation, depicted in Figure 8, would obviously contradict with the observation 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.

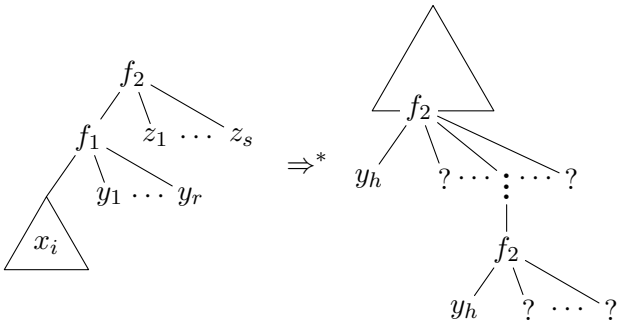

Figure 8. Nested f_2-translations of the same context parameter would violate the uniqueness condition.

4. The full algorithm

Based on the ideas from the previous section we describe the lazy composition algorithm formally by syntax-directed transformation rules.

One detail we have disregarded so far is the fact that not all context parameters of f_1 need to have the same type as the intermediate data being eliminated. Thus, 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 $f_2 \notin$*

[4] This follows because the expression $(f_2\ y_h\ v_{h,1} \cdots v_{h,s})$ in that argument position contains the only appearance of $v_{h,l}$ in the right-hand side of the binding.

$\mathcal{G}$), 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 an 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 $(f_1\ \theta\ \phi_1 \cdots \phi_r)$ [5]. We replace e by the expression

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

where the $c, c_{1,1}, \ldots, c_{r',s}$ are fresh variables and $\overline{f_1 f_2}$ 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 an equation for $\overline{f_1 f_2}$ at the same pattern:

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

where $\mathcal{L}[\![.]\!]$ is defined by the eight rules given below. As a side-effect, $\mathcal{L}[\![.]\!]$ produces further bindings to be collected in the **let***-block.*

1. $\mathcal{L}[\![v]\!] \rightsquigarrow v$.
2. $\mathcal{L}[\![\mathsf{C}\ \xi_1 \cdots \xi_n]\!] \rightsquigarrow \mathsf{C}\ \mathcal{L}[\![\xi_1]\!] \cdots \mathcal{L}[\![\xi_n]\!]$.
3. $\mathcal{L}[\![(\xi_1, \ldots, \xi_n)]\!] \rightsquigarrow (\mathcal{L}[\![\xi_1]\!], \ldots, \mathcal{L}[\![\xi_n]\!])$.
4. $\mathcal{L}[\![g\ \xi_1 \cdots \xi_n]\!] \rightsquigarrow g\ \mathcal{L}[\![\xi_1]\!] \cdots \mathcal{L}[\![\xi_n]\!]$, *if* $g \in \mathcal{G}$.
5. $\mathcal{L}[\![f\ x_i\ \xi_1 \cdots \xi_n]\!] \rightsquigarrow f\ x_i\ \mathcal{L}[\![\xi_1]\!] \cdots \mathcal{L}[\![\xi_n]\!]$, *if* $f \in \{f_1, f_2\}$.
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 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 the bindings $v_1 = \mathcal{L}[\![\psi_1]\!], \ldots, v_s = \mathcal{L}[\![\psi_s]\!]$ are produced.

[5] Due to the Hindley-Milner type system—allowing no polymorphic recursion—these are then the only context parameters of f_1 that might potentially be reached by calls of f_2 in the intermediate result.

7. $\mathcal{L}[\![f_2\ y_h\ \psi_1 \cdots \psi_s]\!] \rightsquigarrow y'_h$,
   and the bindings $c_{h,1} = \mathcal{L}[\![\psi_1]\!], \ldots, c_{h,s} = \mathcal{L}[\![\psi_s]\!]$ *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*

$$(v, v_{1,1}, \ldots, v_{r',s}) = \overline{f_1 f_2}\ x_i\ \phi_1 \cdots \phi_r\ \begin{array}{l}\mathcal{L}[\![\psi_1]\!] \cdots \mathcal{L}[\![\psi_s]\!] \\ \mathcal{L}[\![f_2\ \phi_1\ v_{1,1} \cdots v_{1,s}]\!] \\ \cdots \\ \mathcal{L}[\![f_2\ \phi_{r'}\ v_{r',1} \cdots v_{r',s}]\!]\end{array}$$

   is produced.

If for a $c_{h,l}$ *among the* $c_{1,1}, \ldots, c_{r',s}$ *no binding was produced, then we add the binding* $c_{h,l} = \bot$.

The only case not covered by the above rules is the possibility that for $\mathcal{L}[\![f_2\ (\mathsf{D}\ \phi_1 \cdots \phi_n)\ \psi_1 \cdots \psi_s]\!]$ *there is no defining equation of* f_2 *at data constructor* D *in the program. In this case we are at a position where also the original program would fail (if it would reach that position at all), so we may safely proceed by:*

$\mathcal{L}[\![f_2\ (\mathsf{D}\ \phi_1 \cdots \phi_n)\ \psi_1 \cdots \psi_s]\!]$
$\rightsquigarrow$ *error* "Nonexhaustive patterns in function f_2". $\diamond$

4.1. Termination and correctness

While program transformations based on the unfold/fold-technique [3, 10, 30, 37] generally require considerable effort to ensure termination of the transformation process, the restricted form of input programs for lazy composition prevents such problems.

LEMMA 1. *The lazy composition transformation terminates.*

Proof. We measure an argument ξ of $\mathcal{L}[\![.]\!]$ with the pair of natural numbers (p, q), where p is the maximal size[6] of a recursion argument of a call to f_2 in ξ and q is the size of ξ. With the well-founded order defined by

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

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

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

We do not provide a formal correctness proof for lazy composition. Instead, we substantiate the semantic equivalence of the original expression e and its replacement $\overline{e}$ by justifying each of the transformation rules used to build the defining equations of $\overline{f_1 f_2}$. We then give a short discussion on the relation between the total correctness issue for lazy composition and an existing formal correctness proof for the tree transducer composition technique.

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, formally specified by the rules 1–8 from Transformation 1.

It should be clear that the rules 1–6 of $\mathcal{L}[\![.]\!]$ are locally equivalence-preserving. The abstraction from the $\psi_1, \ldots, \psi_s$ by introduction of fresh variables with appropriate bindings in the unfolding rule 6—which corresponds to case **(a)** from Section 3.2—enables the sharing mechanism of lazy evaluation to avoid duplicated computations in the resulting program in case that ψ is nonlinear 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. At this point we also know the context parameters of f_2 at occurrences of y_h, which are exactly the values that we have to provide in the result tuple for the $c_{h,1}, \ldots, c_{h,s}$-positions. Uniqueness of such values is guaranteed by f_1 being context-linear and f_2 being recursion-linear, as discussed in Sections 3.4 and 3.5.

In rule 8 we apply the function $\overline{f_1 f_2}$ to 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 $\phi_1, \ldots, \phi_{r'}$. These are also computed by the $\overline{f_1 f_2}$-function, yielding a circular definition that will always behave well under lazy evaluation (cf. Sections 3.6 and 3.7).

Sands [29] developed and used *improvement theory* to establish total correctness of program transformations based on local equivalences. Unfortunately, his machinery is not readily applicable to lazy composition because our rules 7 and 8 do not fit into this setting. They necessitate and complement each other, but are in general applied at completely different places in the program. Hence, contrary to Sands' modular approach for pure unfold/fold-transformations, a formal proof for lazy composition would have to take into account the transformation of the whole program at once, much as in our correctness proof for the tree transducer composition technique [36].

In fact, the latter provides a certain kind of justification also for the present technique using circular programs. The crucial point about

total correctness of lazy composition is to establish that runs of the resulting program always terminate, i.e. that no truly circular data dependencies can occur. Essentially the same problem surfaces also in the composition of restricted macro tree transducers, but at transformation time. Namely, since no circular bindings can be introduced in that setting, translations of context parameters of the first transducer with functions of the second transducer are repeatedly nested in the right-hand sides of the resulting transducer to provide for potential dependencies between them. By proving noncircularity of the associated dependency relation, it was shown in [36] that this nesting process can safely be stopped after finitely many steps in order not to create right-hand sides of infinite size. That formal proof corresponds to the informal argument made in Section 3.7 with the help of Figure 8. The key difference is that lazy composition does not break the (only potential) circularity at transformation time, but rather lets it be resolved at runtime of the transformed program by the lazy evaluation mechanism.

5. Practical aspects

We apply Transformation 1 to the example from the introduction and consider post-processing issues. Then we discuss relaxations on the linearity restrictions required by lazy composition, and an alternative use of higher-order functions instead of circular bindings.

Example 6. Consider lazy composition for $f_1 = asc$ and $f_2 = unp$ (with $r = r' = s = 1$). Using Transformation 1, we can replace a call

$$e = (unp\ (asc\ t\ (\mathsf{Num}\ 0))\ \text{“”})$$

in the program by the expression

$$\overline{e} = (\mathbf{let}\ (c, c_{1,1}) = \overline{ascunp}\ t\ (\mathsf{Num}\ 0)\ \text{“”}\ (unp\ (\mathsf{Num}\ 0)\ c_{1,1})\ \mathbf{in}\ c)\,,$$

where the function $\overline{ascunp}$ is defined by the equations to be read off from the following calculations (annotated with the applied rules):

$$\begin{array}{lll}
 & \overline{ascunp}\ (\mathsf{Num}\ x)\ y\ z\ y' & \\
= & \mathbf{let}\ c = \mathcal{L}[\![unp\ (\mathsf{Add}\ y\ (\mathsf{Num}\ x))\ z]\!] & \\
 & \mathbf{in}\ \ (c, c_{1,1}) & \\
\rightsquigarrow & \mathbf{let}\ c\ \ = \mathcal{L}[\![unp\ y\ (\text{'+'} : unp\ (\mathsf{Num}\ x)\ v_1)]\!] & \text{— by 6} \\
 & \quad v_1 = \mathcal{L}[\![z]\!] & \\
 & \mathbf{in}\ \ (c, c_{1,1}) & \\
\rightsquigarrow^2 & \mathbf{let}\ c\ \ = y' & \text{— by 7,1} \\
 & \quad v_1\ \ = z & \\
 & \quad c_{1,1} = \mathcal{L}[\![\text{'+'} : unp\ (\mathsf{Num}\ x)\ v_1]\!] & \\
 & \mathbf{in}\ \ (c, c_{1,1}) &
\end{array}$$

$$
\begin{array}{lll}
\rightsquigarrow^2 & \textbf{let } c = y' & \text{— by 2} \\
& \quad\; v_1 = z & \\
& \quad\; c_{1,1} = {}'+' : \mathcal{L}[\![unp\ (\mathsf{Num}\ x)\ v_1]\!] & \\
& \textbf{in } (c, c_{1,1}) & \\
\rightsquigarrow & \textbf{let } c = y' & \text{— by 6} \\
& \quad\; v_1 = z & \\
& \quad\; c_{1,1} = {}'+' : \mathcal{L}[\![shows\ x\ v_2]\!] & \\
& \quad\; v_2 = \mathcal{L}[\![v_1]\!] & \\
& \textbf{in } (c, c_{1,1}) & \\
\rightsquigarrow^4 & \textbf{let } c = y' & \text{— by 4,1} \\
& \quad\; v_1 = z & \\
& \quad\; c_{1,1} = {}'+' : shows\ x\ v_2 & \\
& \quad\; v_2 = v_1 & \\
& \textbf{in } (c, c_{1,1}) & \\
& & \\
& \overline{ascunp}\ (\mathsf{Add}\ x_1\ x_2)\ y\ z\ y' & \\
= & \textbf{let } c = \mathcal{L}[\![unp\ (asc\ x_1\ (asc\ x_2\ y))\ z]\!] & \\
& \textbf{in } (c, c_{1,1}) & \\
\rightsquigarrow & \textbf{let } c = v_1 & \text{— by 8} \\
& \quad\; (v_1, v_2) = \overline{ascunp}\ x_1\ (asc\ x_2\ y)\ \mathcal{L}[\![z]\!] & \\
& \qquad\qquad\qquad\quad \mathcal{L}[\![unp\ (asc\ x_2\ y)\ v_2]\!] & \\
& \textbf{in } (c, c_{1,1}) & \\
\rightsquigarrow^2 & \textbf{let } c = v_1 & \text{— by 1,8} \\
& \quad\; (v_1, v_2) = \overline{ascunp}\ x_1\ (asc\ x_2\ y)\ z\ v_3 & \\
& \quad\; (v_3, v_4) = \overline{ascunp}\ x_2\ y\ \mathcal{L}[\![v_2]\!]\ \mathcal{L}[\![unp\ y\ v_4]\!] & \\
& \textbf{in } (c, c_{1,1}) & \\
\rightsquigarrow^3 & \textbf{let } c = v_1 & \text{— by 1,7} \\
& \quad\; (v_1, v_2) = \overline{ascunp}\ x_1\ (asc\ x_2\ y)\ z\ v_3 & \\
& \quad\; (v_3, v_4) = \overline{ascunp}\ x_2\ y\ v_2\ y' & \\
& \quad\; c_{1,1} = v_4 & \\
& \textbf{in } (c, c_{1,1}) &
\end{array}
$$

In the resulting program the intermediate result is not used anymore. Although fragments of the intermediate data structure still textually appear in the program—e.g. $(asc\ x_2\ y)$—they will not be evaluated under lazy evaluation and can thus be removed by *useless variable elimination* (cf. the next subsection). $\diamond$

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 ($unp\ (asc\ t\ (\mathsf{Num}\ 0))\ ""$), thus gaining the final program from the introduction.

Firstly, the result of an $\overline{ascunp}$-call never depends 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 $\overline{f_1 f_2}$ it occurs elsewhere than inside the j-th argument positions of recursive calls to $\overline{f_1 f_2}$. A useless argument can be removed from every $\overline{f_1 f_2}$-call and from every defining equation of $\overline{f_1 f_2}$. Doing so for the program in Example 6 gives the definition

$$\begin{array}{lll}
\overline{ascunp}'\ (\mathsf{Num}\ x) & z\ y' = \mathbf{let} & c = y' \\
 & & v_1 = z \\
 & & c_{1,1} = '+' : shows\ x\ v_2 \\
 & & v_2 = v_1 \\
 & \mathbf{in} & (c, c_{1,1}) \\
\overline{ascunp}'\ (\mathsf{Add}\ x_1\ x_2)\ z\ y' = \mathbf{let} & c = v_1 \\
 & & (v_1, v_2) = \overline{ascunp}'\ x_1\ z\ v_3 \\
 & & (v_3, v_4) = \overline{ascunp}'\ x_2\ v_2\ y' \\
 & & c_{1,1} = v_4 \\
 & \mathbf{in} & (c, c_{1,1})
\end{array}$$

to be used with the expression

$$\overline{e}' = (\mathbf{let}\ (c, c_{1,1}) = \overline{ascunp}'\ t\ ""\ (unp\ (\mathsf{Num}\ 0)\ c_{1,1})\ \mathbf{in}\ c).$$

Secondly, some of the **let**-bindings in the above equations can be *inlined*. There are several strategies for inlining [25]; in our example we choose to inline all variables that are not bound as part of a tuple. The result of this inlining and an additional unfold-step in $\overline{e}'$ is then the program of expression $\overline{e}''$ and function $\overline{ascunp}''$ from the introduction.

Considering this program, another optimization becomes possible by realizing that for every terminating call of $\overline{ascunp}''$ the first element of the result tuple is 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* [34] for macro tree transducers. By replacing in the right-hand sides of defining equations for $\overline{ascunp}''$ all constructor applications and calls to other functions than $\overline{ascunp}''$ with the special symbol $\star$, we obtain the following version:

$$\begin{array}{lll}
\overline{ascunp}''\ (\mathsf{Num}\ x) & z\ y' = (y', \star) \\
\overline{ascunp}''\ (\mathsf{Add}\ x_1\ x_2)\ z\ y' = \mathbf{let} & (v_1, v_2) = \overline{ascunp}''\ x_1\ z\ \ v_3 \\
 & \quad (v_3, v_4) = \overline{ascunp}''\ x_2\ v_2\ y' \\
 & \mathbf{in} \quad (v_1, v_4)
\end{array}$$

From the base case $(\mathsf{Num}\ x)$ we obtain the conjecture that $\overline{ascunp}''$ 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 value than just an argument from the left-hand side is returned there. By applying this conjecture for the recursive calls in the inductive case $(\mathsf{Add}\ x_1\ x_2)$, the second equation is transformed into:

$$\begin{array}{rl} \overline{ascunp}''\ (\mathsf{Add}\ x_1\ x_2)\ z\ y' = \textbf{let} & (v_1, v_2) = (v_3, \star) \\ & (v_3, v_4) = (y', \star) \\ \textbf{in} & (v_1, v_4) \end{array}$$

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

Using the thus gained information, we can systematically optimize $\overline{ascunp}''$ further into $\overline{ascunp}'''$ by replacing every binding of the form

$$(v, v_{1,1}) = \overline{ascunp}''\ x\ \psi\ \phi'$$

with the pair of bindings

$$\begin{array}{ll} v & = \phi' \\ v_{1,1} & = \overline{ascunp}'''\ x\ \psi\ v \end{array}$$

and dropping the first element of the result tuple in every defining equation. Thus, we get the expression

$$\begin{array}{rl} \overline{e}''' = (\textbf{let}\ c & = shows\ 0\ c_{1,1} \\ c_{1,1} & = \overline{ascunp}'''\ t\ \text{“”}\ c \\ \textbf{in}\ c) & \end{array}$$

and the following function definition:

$$\begin{array}{lll} \overline{ascunp}'''\ (\mathsf{Num}\ x) & z\ y' = {}'+' : shows\ x\ z & \\ \overline{ascunp}'''\ (\mathsf{Add}\ x_1\ x_2)\ z\ y' = \textbf{let} & v_1 = v_3 & \\ & v_2 = \overline{ascunp}'''\ x_1\ z\ v_1 & \\ & v_3 = y' & \\ & v_4 = \overline{ascunp}'''\ x_2\ v_2\ v_3 & \\ \textbf{in} & v_4 & \end{array}$$

Finally, inlining and removal of the useless last argument of $\overline{ascunp}'''$ yields the expression $\overline{e}''''$ together with the function $\overline{ascunp}''''$ as given in the introduction.

Measurements show that the optimization by elimination of tuple elements is crucial for achieving an efficiency improvement. Hence, it would be desirable to characterize subclasses of programs for which the function produced by lazy composition computes one of its result tuple elements always by projection on one of its parameters. To give one sufficient condition for this, we need the notions of *recursive* and

nonrecursive constructors, respectively. A data constructor is called recursive if at least one of its arguments in the corresponding data type definition is of the type being defined; otherwise, it is called nonrecursive. For example, Node and (:) are recursive constructors, while Tip and [] are nonrecursive.

The function $\overline{f_1 f_2}$ produced by lazy composition will in the first element of its result tuple always project on one of its $y'_1, \ldots, y'_{r'}$-parameters if the right-hand sides of equations for f_1 contain no nonrecursive constructors and contain recursion variables of the output type of f_1 only as first arguments of recursive calls, and if every right-hand side of an equation of f_2 for a recursive constructor is rooted by a call to f_2. If additionally $r' = 1$, then the optimization presented above can eliminate the first element of the result tuple because that one will—detectably—always be equal to the y'_1-argument. Examples fulfilling the condition on f_1 are *pre* and *hanoi*, but not *asc*. Examples fulfilling the condition on f_2 are *rev* and *exp*, but not (++).

Another sufficient condition for the elimination of a tuple element to be applicable is when some context parameter of f_2 is passed unchanged to the same parameter position of all recursive calls, as is the case for (++).

5.2. Changing termination behavior on infinite inputs

The tuple elimination step of the post-processing presented in the previous subsection sometimes transforms nonterminating programs into terminating ones. For illustration, consider the definition

```
infinite_term :: Term
infinite_term = h 1
  where h :: Int → Term
        h n = Add (h (n + 1)) (Num n)
```

of the infinite term depicted in Figure 9. The evaluation of the original expression (*unp* (*asc infinite_term* (Num 0)) "") from the introductory example leads to an infinite reduction never producing any output. In contrast, the expression (*shows* 0 ($\overline{asc\,unp}''''$ *infinite_term* "")) obtained by lazy composition plus post-processing produces the infinite output string "$0 + 1 + 2 + 3 + 4 + \cdots$". When applying Haskell's standard *head* function to the original expression and to the transformed expression, respectively, the former program does not terminate, while the latter does so with result '0'. The change in termination behavior is caused by the fact that the "abstract interpretation" in the previous subsection considered only terminating calls of $\overline{asc\,unp}''$, whereas the outcome of this analysis was applied unconditionally, thus affecting also nonterminating calls of the function.

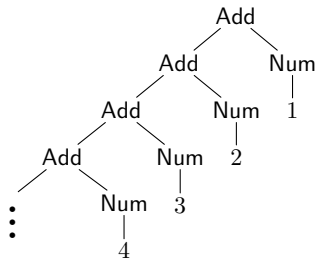

Figure 9. infinite_term

Since it cannot happen that conversely a terminating program is turned into a nonterminating one, we might ignore this phenomenon or even be pleased that the termination behavior can be improved. In some situations—e.g. during testing of programs—such a semantic change is undesirable because a nontermination usually indicates an algorithmic mistake that should not be "hidden" by the compiler. Hence, the user should be given the possibility to control the application of transformations that might turn nonterminating programs into terminating ones, e.g. through a compiler flag.

5.3. Relaxing linearity restrictions

Note that lazy composition as presented in Transformation 1 does not require f_1 to be recursion-linear. Thus, it can also handle functions like *exp* from Figure 3, which cannot be transformed into an attribute grammar of the form required for the deforestation method by attribute grammar composition [5, 16]. In this subsection we discuss how the linearity restrictions on the mtt-functions involved in lazy composition can be relaxed a bit further.

As an example, consider the function *hanoi* from Figure 3, implementing the recursive solution to the well-known "Towers of Hanoi" problem by using an accumulating parameter *ys* to avoid inefficient concatenations. Assume that for a given number of discs d we want to compute the reversed list of moves, i.e. (*rev* (*hanoi* d 1 3 2 []) []).

Note that lazy composition is not immediately applicable because *hanoi* is not context-linear as required for f_1 in Transformation 1. However, we only imposed linearity restrictions in order to avoid two different calls of f_2 on the same context parameter of f_1. Since in the definition of *hanoi* only the first three context parameters are copied and—for typing reasons—these will never be reached by a call of *rev* during the translation of *hanoi*'s right-hand sides, the violation of context-linearity does no harm here. Hence, lazy composition is applicable as follows.

Example 7. Transformation 1 assumes that among the context parameters of f_1 the ones with the same type as f_1's result appear first. Since this is not the case for *hanoi*, a reordering making ys the first context parameter would be necessary (such that then $r' = 1$). For the sake of clarity, we abstain from this reordering in the following calculations:

$$
\begin{array}{lll}
 & \overline{hanrev}\ 0\ y_1\ y_2\ y_3\ ys\ zs\ ys' & \\
= & \textbf{let}\ c = \mathcal{L}[\![rev\ ys\ zs]\!] & \\
 & \textbf{in}\ (c, c_{4,1}) & \\
\leadsto & \textbf{let}\ c = ys' & \text{— by 7} \\
 & \quad c_{4,1} = \mathcal{L}[\![zs]\!] & \\
 & \textbf{in}\ (c, c_{4,1}) & \\
\leadsto & \textbf{let}\ c = ys' & \text{— by 1} \\
 & \quad c_{4,1} = zs & \\
 & \textbf{in}\ (c, c_{4,1}) &
\end{array}
$$

$$
\begin{array}{lll}
 & \overline{hanrev}\ (x+1)\ y_1\ y_2\ y_3\ ys\ zs\ ys' & \\
= & \textbf{let}\ c = \mathcal{L}[\![rev\ (hanoi\ x\ y_1\ y_3\ y_2 & \\
 & \qquad ((y_1, y_2) : (hanoi\ x\ y_3\ y_2\ y_1\ ys)))\ zs]\!] & \\
 & \textbf{in}\ (c, c_{4,1}) & \\
\leadsto & \textbf{let}\ c = v_1 & \text{— by 8} \\
 & \quad (v_1, v_2) = \overline{hanrev}\ x\ y_1\ y_3\ y_2 & \\
 & \qquad ((y_1, y_2) : (hanoi\ x\ y_3\ y_2\ y_1\ ys))\ \mathcal{L}[\![zs]\!] & \\
 & \qquad \mathcal{L}[\![rev\ ((y_1, y_2) : (hanoi\ x\ y_3\ y_2\ y_1\ ys)) & \\
 & \qquad\quad v_2]\!] & \\
 & \textbf{in}\ (c, c_{4,1}) & \\
\leadsto^2 & \textbf{let}\ c = v_1 & \text{— by 1,6} \\
 & \quad (v_1, v_2) = \overline{hanrev}\ x\ y_1\ y_3\ y_2 & \\
 & \qquad ((y_1, y_2) : (hanoi\ x\ y_3\ y_2\ y_1\ ys))\ zs & \\
 & \qquad \mathcal{L}[\![rev\ (hanoi\ x\ y_3\ y_2\ y_1\ ys) & \\
 & \qquad\quad ((y_1, y_2) : v_3)]\!] & \\
 & \quad v_3 = \mathcal{L}[\![v_2]\!] & \\
 & \textbf{in}\ (c, c_{4,1}) & \\
\leadsto^2 & \textbf{let}\ c = v_1 & \text{— by 8,1} \\
 & \quad (v_1, v_2) = \overline{hanrev}\ x\ y_1\ y_3\ y_2 & \\
 & \qquad ((y_1, y_2) : (hanoi\ x\ y_3\ y_2\ y_1\ ys))\ zs\ v_4 & \\
 & \quad v_3 = v_2 & \\
 & \quad (v_4, v_5) = \overline{hanrev}\ x\ y_3\ y_2\ y_1 & \\
 & \qquad ys\ \mathcal{L}[\![((y_1, y_2) : v_3)]\!]\ \mathcal{L}[\![rev\ ys\ v_5]\!] & \\
 & \textbf{in}\ (c, c_{4,1}) &
\end{array}
$$

$$
\begin{array}{lll}
\leadsto^7 \textbf{let } c & = v_1 & \text{— by 2,3,} \\
\quad (v_1, v_2) & = \overline{hanrev}\ x\ y_1\ y_3\ y_2 & \text{1,7} \\
\quad & \quad ((y_1, y_2) : (hanoi\ x\ y_3\ y_2\ y_1\ ys))\ zs\ v_4 & \\
\quad v_3 & = v_2 & \\
\quad (v_4, v_5) & = \overline{hanrev}\ x\ y_3\ y_2\ y_1\ ys\ ((y_1, y_2) : v_3)\ ys' & \\
\quad c_{4,1} & = v_5 & \\
\textbf{in } (c, c_{4,1}) & &
\end{array}
$$

Post-processing as in Section 5.1 transforms the replacement

$$\textbf{let } (c, c_{4,1}) = \overline{hanrev}\ d\ 1\ 3\ 2\ []\ []\ (rev\ []\ c_{4,1})\ \textbf{in } c$$

for

$$rev\ (hanoi\ d\ 1\ 3\ 2\ [])\ []$$

into

$$\overline{hanrev}''''\ d\ 1\ 3\ 2\ [],$$

where the following defining equations are produced:

$$
\begin{array}{lll}
\overline{hanrev}''''\ 0 & y_1\ y_2\ y_3\ zs = zs & \\
\overline{hanrev}''''\ (x+1)\ y_1\ y_2\ y_3\ zs & = \overline{hanrev}''''\ x\ y_3\ y_2\ y_1 & \\
& \quad ((y_1, y_2) : (\overline{hanrev}''''\ x\ y_1\ y_3\ y_2\ zs)) & \diamond
\end{array}
$$

For similar reasons as discussed above the previous example, the restriction on f_2 in the lazy composition transformation can be relaxed 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 Transformation 1.

Further extensions to the applicability of lazy composition have been sketched in the appendix of [35].

5.4. Replacing circularity by higher-orderedness

One referee of the conference version of this paper suggested to eliminate the circularities from our resulting tupled programs by using the lambda-abstraction strategy [22, 23], thus obviating the need for costly lazy evaluation. In this subsection we follow this advice for our introductory example, but also demonstrate its limitations by giving a negative example.

Example 8. Consider the circular program consisting of the expression $\overline{e}''$ and the function definition for $\overline{ascunp}''$ from the introduction.

To perform an *export of information* in the terminology of [23], we first analyze the actual dependencies of the two result tuple elements on the parameters of $\overline{ascunp}''$. This step is similar to the computation of attribute dependencies for an attribute grammar and yields that the y' parameter is only required for the first element and the z parameter only for the second element of the result tuple, i.e. the following equality holds:

$$\overline{ascunp}''\ t\ z\ y' = (fst\ (\overline{ascunp}''\ t\ \bot\ y'), snd\ (\overline{ascunp}''\ t\ z\ \bot)).$$

Hence, if we introduce a variant $\overline{ascunp}^\star$ of $\overline{ascunp}''$ by removing the last two arguments, but parameterizing the tuple elements accordingly with lambda-abstractions:

$$\overline{ascunp}^\star\ t\ = (\lambda y' \to fst\ (\overline{ascunp}''\ t\ \bot\ y'), \lambda z \to snd\ (\overline{ascunp}''\ t\ z\ \bot))\,,$$

then we can replace a binding of the form

$$(c, c_{1,1}) = \overline{ascunp}''\ t\ z\ y'$$

by the triple of bindings

$$\begin{array}{ll} (c^\star, c^\star_{1,1}) = \overline{ascunp}^\star\ t \\ c & = c^\star\ y' \\ c_{1,1} & = c^\star_{1,1}\ z \end{array}$$

for fresh variables $c^\star$ and $c^\star_{1,1}$.

From $\overline{e}''$ this strategy gives the expression

$$\begin{array}{lll} \overline{e}^\star = (\mathbf{let} & (c^\star, c^\star_{1,1}) = \overline{ascunp}^\star\ t \\ & c & = c^\star\ (shows\ 0\ c_{1,1}) \\ & c_{1,1} & = c^\star_{1,1}\ \text{“”} \\ \mathbf{in} & c) \end{array}$$

without any circular dependencies between bindings.

It still remains to derive a new noncircular definition for the function $\overline{ascunp}^\star :: \mathsf{Term} \to (\mathsf{String} \to \mathsf{String}, \mathsf{String} \to \mathsf{String})$, independent of the circular $\overline{ascunp}''$. To do so, we can use the definition of $\overline{ascunp}^\star$ in terms of $\overline{ascunp}''$, the defining equations for $\overline{ascunp}''$, and the above strategy for replacing calls to $\overline{ascunp}''$ with calls to $\overline{ascunp}^\star$.

For the base case we derive the defining equation as follows:

$$\begin{array}{ll} & \overline{ascunp}^\star\ (\mathsf{Num}\ x) \\ = & (\lambda y' \to fst\ (\overline{ascunp}''\ (\mathsf{Num}\ x)\ \bot\ y'), \\ & \ \lambda z \to snd\ (\overline{ascunp}''\ (\mathsf{Num}\ x)\ z\ \bot)) \\ = & (\lambda y' \to fst\ (y', '+' : shows\ x\ \bot), \lambda z \to snd\ (\bot, '+' : shows\ x\ z)) \\ = & (\lambda y' \to y', \lambda z \to '+' : shows\ x\ z) \end{array}$$

For the recursive case we calculate (additionally using **let**-floating [26] and inlining):

$$
\begin{array}{ll}
 & \overline{ascunp}^{\star}\ (\mathsf{Add}\ x_1\ x_2) \\
= & (\lambda y' \to fst\ (\overline{ascunp}''\ (\mathsf{Add}\ x_1\ x_2)\ \bot\ y'), \\
 & \ \lambda z \to snd\ (\overline{ascunp}''\ (\mathsf{Add}\ x_1\ x_2)\ z\ \bot)) \\
= & (\lambda y' \to fst\ (\mathbf{let}\ (v_1, v_2) = \overline{ascunp}''\ x_1\ \bot\ v_3 \\
 & \qquad\qquad\qquad\ \ (v_3, v_4) = \overline{ascunp}''\ x_2\ v_2\ y' \\
 & \qquad\qquad\ \ \mathbf{in}\ \ (v_1, v_4)), \\
 & \ \lambda z \to snd\ (\mathbf{let}\ (v_1, v_2) = \overline{ascunp}''\ x_1\ z\ \ v_3 \\
 & \qquad\qquad\qquad\ \ (v_3, v_4) = \overline{ascunp}''\ x_2\ v_2\ \bot \\
 & \qquad\qquad\ \ \mathbf{in}\ \ (v_1, v_4))) \\
= & (\lambda y' \to fst\ (\mathbf{let}\ (v_1^{\star}, v_2^{\star}) = \overline{ascunp}^{\star}\ x_1 \\
 & \qquad\qquad\qquad\ \ v_1 \qquad\ = v_1^{\star}\ v_3 \\
 & \qquad\qquad\qquad\ \ v_2 \qquad\ = v_2^{\star}\ \bot \\
 & \qquad\qquad\qquad\ \ (v_3^{\star}, v_4^{\star}) = \overline{ascunp}^{\star}\ x_2 \\
 & \qquad\qquad\qquad\ \ v_3 \qquad\ = v_3^{\star}\ y' \\
 & \qquad\qquad\qquad\ \ v_4 \qquad\ = v_4^{\star}\ v_2 \\
 & \qquad\qquad\ \ \mathbf{in}\ \ (v_1, v_4)), \\
 & \ \lambda z \to snd\ (\mathbf{let}\ (v_1^{\star}, v_2^{\star}) = \overline{ascunp}^{\star}\ x_1 \\
 & \qquad\qquad\qquad\ \ v_1 \qquad\ = v_1^{\star}\ v_3 \\
 & \qquad\qquad\qquad\ \ v_2 \qquad\ = v_2^{\star}\ z \\
 & \qquad\qquad\qquad\ \ (v_3^{\star}, v_4^{\star}) = \overline{ascunp}^{\star}\ x_2 \\
 & \qquad\qquad\qquad\ \ v_3 \qquad\ = v_3^{\star}\ \bot \\
 & \qquad\qquad\qquad\ \ v_4 \qquad\ = v_4^{\star}\ v_2 \\
 & \qquad\qquad\ \ \mathbf{in}\ \ (v_1, v_4))) \\
= & \mathbf{let}\ (v_1^{\star}, v_2^{\star}) = \overline{ascunp}^{\star}\ x_1 \\
 & \qquad\ (v_3^{\star}, v_4^{\star}) = \overline{ascunp}^{\star}\ x_2 \\
 & \ \ \mathbf{in}\ \ (\lambda y' \to fst\ (v_1^{\star}\ (v_3^{\star}\ y'), v_4^{\star}\ (v_2^{\star}\ \bot)), \\
 & \qquad\ \lambda z \to snd\ (v_1^{\star}\ (v_3^{\star}\ \bot), v_4^{\star}\ (v_2^{\star}\ z))) \\
= & \mathbf{let}\ (v_1^{\star}, v_2^{\star}) = \overline{ascunp}^{\star}\ x_1 \\
 & \qquad\ (v_3^{\star}, v_4^{\star}) = \overline{ascunp}^{\star}\ x_2 \\
 & \ \ \mathbf{in}\ \ (\lambda y' \to v_1^{\star}\ (v_3^{\star}\ y'), \lambda z \to v_4^{\star}\ (v_2^{\star}\ z)) \quad \diamond
\end{array}
$$

Note that the program finally obtained in the previous example indeed contains no circular definitions and hence terminates even under eager evaluation. When compiled with a standard lazy compiler, however, its performance is rather disappointing (see Table I in Section 7.1).

Moreover, the strategy of replacing circularity by higher-orderedness is not successful in general, as we now demonstrate with an artificial example. Assume given a context-linear function f_1 that—beside some others—has the following two defining equations for nullary construc-

tors N_1 and N_2:

$$f_1\ \mathsf{N}_1\ y_1\ y_2 = \mathsf{Add}\ y_1\ y_2$$
$$f_1\ \mathsf{N}_2\ y_1\ y_2 = \mathsf{Add}\ y_2\ y_1$$

If we apply lazy composition for this f_1 and $f_2 = asc$, then we obtain—after useless variable elimination and inlining—the equations

$$\overline{f_1\, asc}''\ \mathsf{N}_1\ z\ y_1'\ y_2' = (y_1', y_2', z)$$
$$\overline{f_1\, asc}''\ \mathsf{N}_2\ z\ y_1'\ y_2' = (y_2', z, y_1')$$

and for an expression of the form $(asc\ (f_1\ t\ y_1\ y_2)\ z)$ the replacement

$$\textbf{let}\ (c, c_{1,1}, c_{2,1}) = \overline{f_1\, asc}''\ t\ z\ (asc\ y_1\ c_{1,1})\ (asc\ y_2\ c_{2,1})\ \textbf{in}\ c.$$

Dependency analysis based on the defining equations of $\overline{f_1\, asc}''$ yields that the first element of its result tuple requires y_1' and y_2', the second element requires z and y_2', and the third element requires z and y_1'.[7] Now, by analogy with Example 8, a variant $\overline{f_1\, asc}^{\star}$ of $\overline{f_1\, asc}''$ would be introduced and the circular binding in the above expression involving $\overline{f_1\, asc}''$ would have to be adapted to the following:

$$\begin{array}{lll}
\textbf{let} & (c^\star, c^\star_{1,1}, c^\star_{2,1}) = \overline{f_1\, asc}^{\star}\ t \\
& c & = c^\star\ (asc\ y_1\ c_{1,1})\ (asc\ y_2\ c_{2,1}) \\
& c_{1,1} & = c^\star_{1,1}\ z\ (asc\ y_2\ c_{2,1}) \\
& c_{2,1} & = c^\star_{2,1}\ z\ (asc\ y_1\ c_{1,1}) \\
\textbf{in} & c.
\end{array}$$

However, here we have circular definitions yet again, so eager evaluation will not be sufficient.

6. Related work

In this section we compare lazy composition with classical deforestation and shortcut fusion on a qualitative level. We also distinguish the current work from tree transducer composition.

6.1. Classical deforestation

Since mtt-functions may be defined using nesting of terms in context parameter positions and hence are not *treeless* programs in general, Wadler's original deforestation algorithm [37] does not apply directly

[7] Obviously, the other defining equations of f_1 can be chosen in such a way that no more dependencies than these occur.

to them. This problem can be solved by abstracting context parameters using **let**-expressions explicitly [10] or implicitly [17], which yields an algorithm along the lines of Section 3.1.

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 plus post-processing, respectively.

Example 9. Consider the deforested program from Example 2. Lazy evaluation for the input (Add (Num 1) (Num 2)) yields the computation

$$\begin{aligned}
&\overline{ascunp}\ (\mathsf{Add}\ (\mathsf{Num}\ 1)\ (\mathsf{Num}\ 2))\ \underline{(\mathsf{Num}\ 0)}\ \text{“”}\\
\Rightarrow\ &\overline{ascunp}\ (\mathsf{Num}\ 1)\ (asc\ (\mathsf{Num}\ 2)\ \underline{(\mathsf{Num}\ 0)})\ \text{“”}\\
\Rightarrow\ &unp\ (asc\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 0))\ ('+' : shows\ 1\ \text{“”})\\
\Rightarrow\ &unp\ \underline{(\mathsf{Add}\ (\mathsf{Num}\ 0)\ (\mathsf{Num}\ 2))}\ ('+' : shows\ 1\ \text{“”})\\
\Rightarrow\ &unp\ \underline{(\mathsf{Num}\ 0)}\ ('+' : unp\ \underline{(\mathsf{Num}\ 2)}\ ('+' : shows\ 1\ \text{“”}))\\
\Rightarrow\ &shows\ 0\ ('+' : unp\ \underline{(\mathsf{Num}\ 2)}\ ('+' : shows\ 1\ \text{“”}))\\
\Rightarrow\ &shows\ 0\ ('+' : shows\ 2\ ('+' : shows\ 1\ \text{“”}))\,,
\end{aligned}$$

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 plus post-processing:

$$\begin{aligned}
&shows\ 0\ (\overline{ascunp}''''\ (\mathsf{Add}\ (\mathsf{Num}\ 1)\ (\mathsf{Num}\ 2))\ \text{“”})\\
\Rightarrow\ &shows\ 0\ (\overline{ascunp}''''\ (\mathsf{Num}\ 2)\ (\overline{ascunp}''''\ (\mathsf{Num}\ 1)\ \text{“”}))\\
\Rightarrow\ &shows\ 0\ ('+' : shows\ 2\ (\overline{ascunp}''''\ (\mathsf{Num}\ 1)\ \text{“”}))\\
\Rightarrow\ &shows\ 0\ ('+' : shows\ 2\ ('+' : shows\ 1\ \text{“”}))\,.\quad \diamond
\end{aligned}$$

On the other hand, variants of classical deforestation are applicable to a bigger class of programs than just mtt-functions.

6.2. Shortcut fusion

Shortcut fusion achieves elimination of intermediate results by using higher-order, polymorphic combinators, compositions of which can be transformed by *cata*/*build*-rules [8, 9, 12]. Such rules embody the idea of substituting the operations that are to replace the data constructors of an intermediate result directly into the algorithm producing it, put forward also in [32].

To this aim, the consumer of an intermediate result needs to be expressed as a *catamorphism*. Such a representation can be synthesized

from the defining equations of an mtt-function in a systematic way, using a higher-order catamorphism to capture context parameters.

Example 10. Using the catamorphism combinator

$$\begin{aligned}
&cata_{\mathsf{Term}} :: (\mathsf{Int} \to \alpha) \to (\alpha \to \alpha \to \alpha) \to \mathsf{Term} \to \alpha \\
&cata_{\mathsf{Term}}\ n\ a\ (\mathsf{Num}\ x) \quad\ = n\ x \\
&cata_{\mathsf{Term}}\ n\ a\ (\mathsf{Add}\ x_1\ x_2) \ = a\ (cata_{\mathsf{Term}}\ n\ a\ x_1)\ (cata_{\mathsf{Term}}\ n\ a\ x_2)
\end{aligned}$$

the consumer *unp* from the introductory example can be expressed as

$$unp = cata_{\mathsf{Term}}\ (\lambda x\ z \to shows\ x\ z)\ (\lambda r_1\ r_2\ z \to r_1\ ('+' : r_2\ z)). \quad \diamond$$

On the other hand, all data constructors need to be abstracted uniformly from the producer of an intermediate result in a polymorphic way. Since in the case of mtt-functions parts of the produced output can be "hidden" in the context parameters, these have to be prepared for abstraction via an additional traversal.

Example 11. Using the *rank-2 polymorphic* combinator

$$\begin{aligned}
&build_{\mathsf{Term}} :: (\forall\alpha.\ (\mathsf{Int} \to \alpha) \to (\alpha \to \alpha \to \alpha) \to \alpha) \to \mathsf{Term} \\
&build_{\mathsf{Term}}\ g = g\ \mathsf{Num}\ \mathsf{Add}
\end{aligned}$$

the producer *asc* from the introductory example can be expressed as

$$\begin{aligned}
asc\ t\ y = build_{\mathsf{Term}}\ (\lambda n\ a \to\ &\mathbf{let}\ h\ (\mathsf{Num}\ x) \quad\ y = a\ y\ (n\ x) \\
&\quad\ \ h\ (\mathsf{Add}\ x_1\ x_2)\ y = h\ x_1\ (h\ x_2\ y) \\
&\mathbf{in}\ \ h\ t\ (cata_{\mathsf{Term}}\ n\ a\ y)).
\end{aligned}$$

Now, the rule

$$cata_{\mathsf{Term}}\ n\ a\ (build_{\mathsf{Term}}\ g) = g\ n\ a$$

can be applied to the expression $e = (unp\ (asc\ t\ (\mathsf{Num}\ 0))\ \text{“”})$. After some beta-reductions and an unfolding, the following results:

$$\begin{aligned}
(&\mathbf{let}\ h\ (\mathsf{Num}\ x) \quad\ y = \lambda z \to y\ ('+' : shows\ x\ z) \\
&\quad\ \ h\ (\mathsf{Add}\ x_1\ x_2)\ y = h\ x_1\ (h\ x_2\ y) \\
&\mathbf{in}\ \ h\ t\ (\lambda z \to shows\ 0\ z))\ \text{“”}.
\end{aligned}$$

Here the intermediate term has been eliminated, but in its stead a sequence of suspended function calls has been introduced. An equivalent result could have been obtained by using a variant of Johann's *augment*-combinator together with its associated *cata*/*augment*-rule [12]. $\diamond$

The observed introduction of suspended function calls compromises the intended efficiency gain by shortcut fusion, as discussed by Svenningsson [31] who for this reason disputes the use of higher-order catamorphisms like above. As remedy he proposes the *destroy*/*unfoldr*-rule

(for the list case), which however handles accumulating parameters only for *consumers* of intermediate lists; the more difficult problem of *producers* using accumulating parameters is not approached by his rule. Interestingly though, it successfully handles consumers recursing over several lists simultaneously, which is not addressed by our technique.

Nishimura [20] uses type information to guide a higher-order removal phase that transforms programs resulting from shortcut fusion into tupled, circular programs similar to the results of lazy composition. He informally compares his technique with our work on lazy composition and tree transducer composition [35, 36] and observes that it matches the transformational power of our method quite closely.

6.3. Tree transducer composition

Kühnemann [16, 17] first proposed to use composition techniques for various classes of tree transducers [6, 7] as a means to eliminate intermediate results in functional programs. In [36] a single composition construction is given that generalizes the previously considered ones. The programs handled by it differ from mtt-functions in that no external calls are considered and recursion variables may not occur elsewhere than as first arguments of recursive calls. However, in contrast to the situation with lazy composition in its current form, also mutually recursive functions can be handled by the construction in [36]. It shares with lazy composition the ideas of "translating" the right-hand sides of the producer of an intermediate result by using the consumer's defining equations, and of holding available translated versions of the producer's accumulating parameters in the composed program. An analogous question to the one raised in Figure 6 then gives rise to conditions on the involved functions that are similar to the linearity restrictions introduced for lazy composition in Section 3.4, albeit in the more general setting of mutual recursion.

The key difference between the construction in [36] and lazy composition lies in how the requested context parameter values of the consumer—needed to make the idea of "providing translated versions of the producer's context parameters" effective—are actually obtained. While the former incorporates an elaborate scheme to construct defining equations of additional functions that compute the required information (in additional traversals of the input data structure), the latter simply passes this information up (as part of a tupled function result) from a unique call site during the translation process. The circular bindings created to access such information necessitate a lazy evaluation mechanism to be used for the programs obtained by lazy composition,

whereas the macro tree transducers produced by the technique from [36] are terminating even under eager evaluation.

One topic of interest in the theoretical study of tree transducer composition is how additional restrictions on the composition partners carry over to the composed program. To perform such a study for lazy composition and the linearity restrictions introduced in Section 2 is complicated by rule 8 from Transformation 1, which duplicates parts of $\mathsf{rhs}_{f_1,\mathsf{C}}$, namely the $\phi_1, \ldots, \phi_{r'}$. This duplication is necessary because the function $\overline{f_1 f_2}$ in general needs to preserve the original context parameters of f_1 (beside their translations with f_2). This need, in turn, arises from the freedom of mtt-functions to use recursion variables elsewhere than as first arguments of recursive calls. If this were outlawed, then it could not anymore be the case that in the original program parts of the intermediate result produced by f_1 are copied literally into the final output by f_2. Hence, in the composed program $\overline{f_1 f_2}$ would not need its $y_1, \ldots, y_r$-parameters, simplifying the expression $\overline{e}$ and rule 8 from Transformation 1 accordingly. Under this assumption and the additional assumptions that f_1 is context-linear and f_2 is recursion-linear, it is easy to see that then for recursion-linear f_1 the right-hand side of each defining equation of $\overline{f_1 f_2}$ contains every recursion variable at most once, and for context-linear f_2 the right-hand side of each defining equation of $\overline{f_1 f_2}$ contains every context variable at most once.

7. 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, such an elimination alone is no guarantee for a better efficiency. The dependency of a circular 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 Section 5.1 can eliminate such additional ballast. When post-processing is not applicable or does not produce satisfactory results, there are also possibilities for run-time improvements using a compiler that generates more efficient code for functions returning tupled results. Van Groningen [33] 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 times 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 latter 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).

Another approach to reduce the runtime costs originating from the building of unnecessary closures could be to investigate how to extend the lazy composition algorithm with a "strictness-guidance", as performed for the tupling transformation strategy by Chin *et al.* [4].

7.1. Measurements

To demonstrate efficiency gains realized by our technique in practice, we perform measurements for three examples. For each example we compare execution times for multiple runs of different program versions with varying input sizes. The program versions considered are: (i) the original program, (ii) the program obtained by applying lazy composition, (iii) the program obtained from (ii) by additionally applying post-processing as discussed in Section 5.1, (iv) the program obtained from the original one by applying classical deforestation as indicated in Section 6.1, and (v) the program obtained by applying shortcut fusion as discussed in Section 6.2. For the introductory example we additionally measure runtimes of the variant obtained in Example 8 by replacing circularity with higher-orderedness.

The different program versions were coded as ordinary Haskell source (available at *http://wwwtcs.inf.tu-dresden.de/~voigt/hosc-measure.lhs*), compiled with the Glasgow Haskell Compiler (version 5.04.1, optimization level `-O`), and run on a Sun Ultra 10 workstation (300MHz, 256MB). The runtimes (in seconds) shown in the measurement tables below are split into the time spent for actual expression evaluation (the first summand) and the time spent on garbage collection (the second summand), as obtained from the statistics produced using the runtime system option `-s`. The given execution times include the test frame with generation of input data and consumption of final output. This is unavoidable because a more detailed cost center profiling would corrupt the precision of the measured garbage collection times considerably.

Table I contains the measurements for the introductory example (*unp* (*asc t* (Num 0)) "") on fully balanced terms of different heights h.

They show no significant improvement in runtime behavior for the programs obtained by classical deforestation (cf. Example 2) and shortcut fusion (cf. Example 11). On the other hand, the program consisting of the expression $\overline{e}''''$ and function $\overline{ascunp}''''$ from the introduction achieves a decrease of the time needed for expression evaluation (mutator time) and a considerable improvement of the time spent on garbage collection—especially for large inputs—and hence of the total runtimes. The higher-order program produced in Example 8 seems to suffer from a high garbage collection expense due to the creation of function closures. Apparently, the compiler could not really profit from the fact that eager evaluation is sufficient for this program in contrast to a circular one. By using a strict pair constructor and manually adding strictness annotations the total runtimes of the higher-order program version can be reduced considerably, but not to the degree of compensating all the overhead compared to the original program.

Table I. (*unp* (*asc t* (Num 0)) “”), n runs with fully balanced terms of height h

$n \times h$:	60000×5	2000×10	60×15	2×20
original progr.	2.0+0.1=2.1	2.0+0.1=2.1	2.0+3.5= 5.5	2.3+ 7.4= 9.7
lazy compos.	3.8+0.1=3.9	3.9+0.6=4.5	3.7+3.9= 7.6	3.8+ 6.9=10.7
+post-process.	1.9+0.0=1.9	1.8+0.0=1.8	1.8+1.4= 3.2	1.9+ 1.6= 3.5
class. deforest.	2.1+0.0=2.1	2.1+0.1=2.2	2.0+3.5= 5.5	2.4+ 7.2= 9.6
shortcut fusion	2.0+0.1=2.1	2.0+0.2=2.2	2.1+3.4= 5.5	2.3+ 7.3= 9.6
$\overline{e}^\star$ and $\overline{ascunp}^\star$	3.3+0.1=3.4	3.6+0.6=4.2	3.7+8.0=11.7	3.8+11.8=15.6

A similar improvement by lazy composition plus post-processing as above is obtained for the example (*rev* (*hanoi d* 1 3 2 []) []) from Section 5.3, as witnessed by the measurements in Table II. Classical deforestation achieves no elimination of any part of the intermediate result in this program because *hanoi* produces its output exclusively inside its accumulating parameter. The performance of the program produced by shortcut fusion is about on a par with that of the program produced by our techniques for relatively small input numbers, but for larger inputs the fused program has a considerably higher garbage collection overhead, probably caused by the introduction of a sequence of suspended function calls as discussed by Svenningsson [31].

Table III shows execution times for (*rev* (*pre t* []) []), which was the running example of the conference version of the present paper [35], where also the differently transformed program versions used for the measurements can be found. Interestingly, on large inputs—where the garbage collection times become dominant—there is an efficiency im-

Table II. (*rev* (*hanoi d* 1 3 2 []) []), n runs with input d

$n \times d$:	30000×5	1000×10	30×15	1×20
original progr.	1.6+0.0=1.6	1.7+0.2=1.9	1.7+3.0=4.7	1.8+6.1=7.9
lazy compos.	3.3+0.2=3.5	3.5+1.2=4.7	3.1+3.6=6.7	3.4+5.2=8.6
+post-process.	1.4+0.0=1.4	1.5+0.0=1.5	1.5+1.2=2.7	1.6+1.3=2.9
class. deforest.	1.6+0.0=1.6	1.7+0.2=1.9	1.6+3.1=4.7	1.8+6.1=7.9
shortcut fusion	1.3+0.0=1.3	1.4+0.1=1.5	1.4+3.0=4.4	1.5+5.1=6.6

provement even for the program obtained by lazy composition only (without post-processing).

Table III. (*rev* (*pre t* []) []), n runs with fully balanced trees of height h

$n \times h$:	100000×5	3000×10	100×15	3×20
original progr.	3.0+0.0=3.0	2.8+0.2=3.0	3.2+5.2= 8.4	3.1+10.6=13.7
lazy compos.	5.0+0.1=5.1	5.2+0.7=5.9	4.7+6.0=10.7	4.6+ 7.5=12.1
+post-process.	2.1+0.0=2.1	2.0+0.0=2.0	2.1+1.8= 3.9	2.0+ 1.7= 3.7
class. deforest.	3.0+0.1=3.1	2.8+0.2=3.0	3.0+5.3= 8.3	3.0+10.7=13.7
shortcut fusion	2.0+0.0=2.0	1.8+0.1=1.9	2.3+3.8= 6.1	2.0+ 6.7= 8.7

8. Conclusion

We have developed a program transformation technique for certain recursive functions with accumulating parameters that eliminates intermediate data structures in compositions of two such functions. To assess better the utility for practical implementations, sufficient conditions should be studied under which lazy composition plus post-processing guarantees an efficiency improvement.

The composition and post-processing constructions for macro tree transducers [34, 36] have been implemented in the Haskell$^+$ program transformation system [11]. As another road to application an analysis phase detecting tree transducers in Haskell source programs and a simple composition transformation have been integrated into a full-scale functional compiler [27]. We want to include lazy composition and related techniques in either of these two frameworks.

Acknowledgements

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. Armin Kühnemann was a competent discussion partner regarding good examples. Thanks are also due to several anonymous reviewers for useful comments and suggestions.

References

1. Bird, R.: 1984, 'Using circular programs to eliminate multiple traversals of data'. *Acta Informatica* **21**(3), 239–250.
2. Burstall, R. and J. Darlington: 1977, 'A transformation system for developing recursive programs'. *J. ACM* **24**(1), 44–67.
3. Chin, W.: 1994, 'Safe fusion of functional expressions II: Further improvements'. *J. Funct. Prog.* **4**(4), 515–555.
4. Chin, W., A. Goh, and S. Khoo: 1999, 'Effective optimisation of multiple traversals in lazy languages'. In: *Partial Evaluation and Semantics-Based Program Manipulation, San Antonio, Texas, Proceedings*. pp. 119–130.
5. Correnson, L., E. Duris, D. Parigot, and G. Roussel: 1999, 'Declarative program transformation: A deforestation case-study'. In: *Principles and Practice of Declarative Programming, Paris, France, Proceedings*, Vol. 1702 of *LNCS*. pp. 360–377.
6. Engelfriet, J. and H. Vogler: 1985, 'Macro tree transducers'. *J. Comput. Syst. Sci.* **31**(1), 71–146.
7. Fülöp, Z.: 1981, 'On attributed tree transducers'. *Acta Cybernetica* **5**, 261–279.
8. Gill, A.: 1996, 'Cheap deforestation for non-strict functional languages'. Ph.D. thesis, University of Glasgow.
9. Gill, A., J. Launchbury, and S. Peyton Jones: 1993, 'A short cut to deforestation'. In: *Functional Programming Languages and Computer Architecture, Copenhagen, Denmark, Proceedings*. pp. 223–232.
10. Hamilton, G. and S. Jones: 1992, 'Extending deforestation for first order functional programs'. In: *1991 Glasgow Workshop on Functional Programming, Portree, Scotland, Proceedings*. pp. 134–145.
11. Höff, M., R. Vater, A. Maletti, A. Kühnemann, and J. Voigtländer: 2001, 'Tree transducer based program transformations for Haskell$^+$'. Progress report, Dresden University of Technology.
http://wwwtcs.inf.tu-dresden.de/∼voigt/hpgreport.ps.gz .
12. Johann, P.: 2002, 'A generalization of short-cut fusion and its correctness proof'. *Higher-Order and Symb. Comp.* **15**(4), 273–300.
13. Johnsson, T.: 1987, 'Attribute grammars as a functional programming paradigm'. In: *Functional Programming Languages and Computer Architecture, Portland, Oregon, Proceedings*, Vol. 274 of *LNCS*. pp. 154–173.
14. Kakehi, K., R. Glück, and Y. Futamura: 2001, 'On deforesting parameters of accumulating maps'. In: *Logic Based Program Synthesis and Transformation, Paphos, Cyprus, Proceedings*, Vol. 2372 of *LNCS*. pp. 46–56.

15. Knuth, D.: 1968, 'Semantics of context-free languages'. *Math. Syst. Theory* **2**(2), 127–145. Corrections 1971, *Ibid.* **5**(1), 95–96.
16. Kühnemann, A.: 1998, 'Benefits of tree transducers for optimizing functional programs'. In: *Foundations of Software Technology & Theoretical Computer Science, Chennai, India, Proceedings*, Vol. 1530 of *LNCS*. pp. 146–157.
17. Kühnemann, A.: 1999, 'Comparison of deforestation techniques for functional programs and for tree transducers'. In: *Functional and Logic Programming, Tsukuba, Japan, Proceedings*, Vol. 1722 of *LNCS*. pp. 114–130.
18. Kuiper, M. and S. Swierstra: 1987, 'Using attribute grammars to derive efficient functional programs'. In: *Computing Science in the Netherlands, Amsterdam, The Netherlands, Proceedings*. pp. 39–52.
19. Milner, R.: 1978, 'A theory of type polymorphism in programming'. *J. Comput. Syst. Sci.* **17**(3), 348–375.
20. Nishimura, S.: 2002, 'Deforesting in accumulating parameters via type-directed transformations'. In: *Asian Workshop on Programming Languages and Systems, Shanghai, China, Proceedings*.
21. Pettorossi, A.: 1977, 'Transformation of programs and use of tupling strategy'. In: *Informatica, Bled, Yugoslavia, Proceedings*. pp. 1–6.
22. Pettorossi, A.: 1987, 'Derivation of programs which traverse their input data only once'. In: *Advanced School on Programming Methodologies, Rome, Italy, Proceedings*. pp. 165–184.
23. Pettorossi, A. and M. Proietti: 1987, 'Importing and exporting information in program development'. In: *Partial Evaluation and Mixed Computation, Gammel Avernaes, Denmark, Proceedings*. pp. 405–425.
24. Peyton Jones, S. (ed.): 2003, *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press.
25. Peyton Jones, S. and S. Marlow: 2002, 'Secrets of the Glasgow Haskell Compiler inliner'. *J. Funct. Prog.* **12**(4/5), 393–433.
26. Peyton Jones, S., W. Partain, and A. Santos: 1996, 'Let-floating: Moving bindings to give faster programs'. In: *International Conference on Functional Programming, Philadelphia, Pennsylvania, Proceedings*. pp. 1–12.
27. Reuther, S.: 2002, 'Adding a tree transducer recognition/transformation pass to the Glasgow Haskell Compiler'. Students project, Dresden University of Technology.
28. Runciman, C., M. Firth, and N. Jagger: 1989, 'Transformation in a non-strict language: An approach to instantiation'. In: *1989 Glasgow Workshop on Functional Programming, Fraserburgh, Scotland, Proceedings*. pp. 133–141.
29. Sands, D.: 1996, 'Total correctness by local improvement in the transformation of functional programs'. *ACM Trans. Prog. Lang. Syst.* **18**(2), 175–234.
30. Sørensen, M., R. Glück, and N. Jones: 1996, 'A positive supercompiler'. *J. Funct. Prog.* **6**(6), 811–838.
31. Svenningsson, J.: 2002, 'Shortcut fusion for accumulating parameters & zip-like functions'. In: *International Conference on Functional Programming, Pittsburgh, Pennsylvania, Proceedings*. pp. 124–132.
32. Swierstra, S. and O. de Moor: 1993, 'Virtual data structures'. In: *Formal Program Development*, Vol. 755 of *LNCS*. pp. 355–371.
33. van Groningen, J.: 1999, 'Optimising recursive functions yielding multiple results in tuples in a lazy functional language'. In: *Implementation of Functional Languages, Lochem, The Netherlands, Proceedings*, Vol. 1868 of *LNCS*. pp. 59–76.

34. Voigtländer, J.: 2001, 'Composition of restricted macro tree transducers'. Master's thesis, Dresden University of Technology.
http://wwwtcs.inf.tu-dresden.de/∼voigt/mscthesis.ps.gz .
35. Voigtländer, J.: 2002, 'Using circular programs to deforest in accumulating parameters'. In: *Asian Symposium on Partial Evaluation and Semantics-Based Program Manipulation, Aizu, Japan, Proceedings*. pp. 126–137.
36. Voigtländer, J. and A. Kühnemann: 2001, 'Composition of functions with accumulating parameters'. Technical Report TUD-FI01-08, Dresden University of Technology. *http://wwwtcs.inf.tu-dresden.de/∼voigt/TUD-FI01-08.ps.gz* .
Revised version to appear in *J. Funct. Prog.*
37. Wadler, P.: 1990, 'Deforestation: Transforming programs to eliminate trees'. *Theor. Comput. Sci.* **73**(2), 231–248.