Deforesting in Accumulating Parameters via Type-Directed Transformations *

Susumu Nishimura

RIMS, Kyoto University
Sakyo-ku 606–8502, JAPAN
nisimura@kurims.kyoto-u.ac.jp

Abstract

Classical deforestation methods fail to compose functions with accumulating parameters, which are function arguments to which intermediate result accumulates. This paper proposes a new deforestation method which solves this problem. The method is comprised of a few transformation steps, each of which is guided by type information. Though our solution draws its fundamental idea from a deforestation method that has been developed for attribute grammars, it is not merely a recast of an existing technique to another formalism. The new method provides a type-based account for the existing attribute grammar deforestation process. Further, it can deforest a class of functions that the attribute grammar deforestation cannot. These advantages are demonstrated by a few examples. It is also shown that the new method is as powerful as another method that has been developed for macro tree transducer composition.

1 Introduction

In functional programming languages, it is common to define a function by a composition of two functions, one of which produces an intermediate data structure that is immediately consumed by the other function. This compositional style of programming promotes the desirable modular program development, but on the other hand inefficiency is caused by the temporary production of the intermediate data.

To remedy this deficiency of functional programming, Wadler [22] proposed a program transformation technique, called *deforestation*, which automatically derives a single function definition that does not produce intermediate data structure, from a given composition of functions.

The deforestation technique, however, cannot be applied to compose functions which have *accumulating parameters*. Accumulating parameters are arguments to which a recursive function accumulates intermediate results. For example, an efficient list reverse function `rev` is defined (in Haskell notation) as below, with an accumulating parameter `h`.

```
rev (Cons(n,l)) h = rev l (Cons(n,h))
rev Nil h = h
```

When applied to such a function, Wadler's deforestation process gets stuck because of the well-known problem of not reaching accumulating results [3]. There are several proposals of refinements and improvements to the deforestation technique [9, 15, 3, 17, 16, 12], but none of them provides a satisfactory solution for deforesting in accumulating parameters.

This paper proposes a type-directed transformation method that aims at deforesting in accumulating parameters. The transformation method we propose is based on a composition method for attribute grammars [13], called *descriptional composition* [7]. This attribute grammar composition method has a strong advantage over the above mentioned deforestation methods that it can compose attribute grammars which have so-called inherited attributes (the counterpart of accumulating parameters in functional languages) under certain conditions.

In spite of this advantage, the attribute grammar composition technique has not been widely recognized. This is because of a few weaknesses that reside in the attribute grammar formalism. First, the description of the composition algorithm is rather complicated and is hard to understand for non-experts. Second, one is forced to write programs in the attribute grammar syntax, which is not a common practice nowadays. Lastly, the class of

*To be presented at the Asian Workshop on Programming Languages and Systems 2002 (APLAS'02).

programs that can be expressed by attribute grammars is rather small and the class of composable programs by the attribute grammar composition method is further limited (see Section 4 for details).

This paper is intended to fix the above problems by incorporating the attribute grammar composition method into the functional programming paradigm. The major contribution of this paper is twofold.

First, our method provides a clear account for the attribute grammar composition process from the functional programming side. The composition process is divided into a few smaller transformation steps, each of which is easier to understand than the whole transformation. Further, types serve as a superior guide that directs each subtransformation a right way to go.

Second, our deforestation method is able to compose more programs than the attribute grammar composition method is. Our deforestation method is formulated in the framework of functional programming, whose high expressiveness makes it possible to reveal the hidden power of the attribute grammar composition method behind the limitation of the attribute grammar formalism.

Recently, a powerful deforestation method has been proposed by Voigtländer and Kühnemann [18, 20, 19]. Their method is based on the framework of *macro tree transducers* [5], which are a close relative of attribute grammars. Their composition method substantially extends the class of composable programs than the attribute grammar composition method. An evidence shows that their method is as powerful as ours, as we will discuss in Section 4.

Related work

Our transformation method is closer to shortcut fusion [9], which is another deforestation method. Shortcut fusion can deforest a large class of programs by a local transformation rule for `foldr`/`build` consumer-producer pairs. This distinguishes shortcut fusion from Wadler's deforestation, which is based on the general folding/unfolding transformation [2]. Our transformation method is not driven by the folding/unfolding transformation either, but by a small set of transformation rules (including both local and non-local ones). Hence our method is closer to shortcut fusion.

Several researchers have tackled the problem of deforesting functions with accumulating parameters. Sheard and Fegaras [15] proposed second-order fusion to deforest higher-order functions. However, their deforestation method requires a post-processing that assumes a particular algebraic property of programs, which is not easily mechanizable. Svenningsson proposed a fusion rule to cancel `destroy`/`unfoldr` pairs, which are dual of `foldr`/`build` pairs in shortcut fusion [16]. However, his method can only deal with accumulating parameters in consumer functions.

In connection to attribute grammars, Correnson et. al. [4] utilized the attribute grammar composition method in order to deforest functional programs, by a translation of functional programs into attribute grammars. This however indicates that the ability of their deforestation method is limited by that of attribute grammars. Kakehi et. al. [12] proposed a fusion method for generalized `map` functions that capture a few features of the attribute grammar composition method. In contrast, ours covers the full features of it.

To the best of our knowledge, the above mentioned macro tree transducer composition method by Voigtländer and Kühnemann appears to be the only one which is as powerful as ours in its ability of deforestation. However, their deforestation strategy is quite different from ours. Theirs is closer to the classical deforestation method in the sense that the program transformation is driven by the folding/unfolding transformation. In contrast, ours is closer to shortcut fusion, as we mentioned earlier. Furthermore, ours is characterized by the extensive use of type information that directs the whole program transformation process.

Outline

The rest of this paper is organized as follows. Section 2 gives a complete example of our deforestation method. Section 3 formalizes our method. Section 4 compares our deforestation method with other methods for attribute grammars and for macro tree transducers. Finally, Section 5 concludes the paper with a few future research topics.

2 Deforestation by Example

This section demonstrates a complete deforestation process by an example.

The example is written in Haskell notation [10] extended with records and variants. A record is a labeled product, written $(l_1 = M_1, \cdots, l_n = M_n)(n \geq 0)$ and a variant is a disjoint sum injected by a label, written $\langle l = M \rangle$. A field selection on a record M at the field

l is written $\#l\ M$, and a case expression

$$\texttt{case}\ M\ \texttt{of}\ \langle l_1 = x_1\rangle \rightarrow M_1;\ldots;\langle l_1 = x_n\rangle \rightarrow M_n$$

selects an appropriate clause depending on the injection label of variant M evaluates to.

For the sake of readability, we may use standard pattern matching notations on records, say, $\backslash(h = x) \rightarrow x$ for a syntactic shorthand of $\backslash x \rightarrow \#h\ x$.

The notation for types are standard. We write $(l_1 :: \tau_1, \cdots, l_n :: \tau_n)$ for a record type whose set of field labels are $l_1, \ldots, l_n$ and $\tau_1, \ldots, \tau_n$ are associated field types. Similarly, we write $\langle l_1 :: \tau_1, \cdots, l_n :: \tau_n\rangle$ for a variant type, where $l_1, \cdots, l_n$ are possible injection labels. In this section, we assume the standard (monomorphic) typing rules for records and variants. For technical reasons, however, we in fact need a non-standard typing rules for variants; See Section 3 for details.

2.1 The source program

The example program considered in this section is given in Figure 1. The function `flat` takes a binary tree representation (specified by the algebraic data type `T`) and outputs an integer list (the algebraic type `L`) of leaf values generated by two repeated pre-order traversals over the binary tree. The function accomplishes the couple of traversals by a single recursive call to the input tree, using the circular **`let`** definition [1]. The function `rev` is the standard list reverse function with an accumulating parameter. The goal of transformation is to deforest `revflat`, the composition of `flat` followed by `rev`.

In the program, each recursive function is defined by using records. For example, the `rev` function definition in Introduction has type $\mathtt{L} \rightarrow \mathtt{L} \rightarrow \mathtt{L}$, but it is redefined as a function of type $\mathtt{L} \rightarrow (\mathtt{h} :: \mathtt{L}) \rightarrow (\mathtt{rev} :: \mathtt{L})$. Note that the use of records does not mean that all the field values in a record to be evaluated simultaneously. Due to Haskell's lazy semantics, they are computed independently on demand.

This redefinition is aimed at having the structure of the source program closer to that of attribute grammars: The set of inputs and the set of outputs (the former corresponds to the so-called inherited attributes and the latter to the so-called synthesized attributes [11]) are made explicit and each input/output value is uniquely identified by a label. This makes it possible to incorporate the idea of the attribute grammar composition method into the functional programming paradigm. Also, record types play a significant role in the proceeding transformation procedure.

2.2 Shortcut fusion applied

Our deforestation process starts with an application of the well-known shortcut fusion transformation [9, 17].

To apply the shortcut fusion rule, we define `flat'`, which is another version of `flat` that abstracts over constructor functions, and `foldr`/`build` operators for lists.

```
flat' ::
  T → (Integer → a → a) → a
    → (l:: a, j :: a) → (fl:: a, fj ::a)
flat' (Node(t1,t2)) =
  \c nl (l=l, j=j) ->
    let syn1 = flat' t1 c nl (l=l, j=j)
        syn2 = flat' t2 c nl
                (l=#fl syn1, j=#fj syn1)
  in (fl=#fl syn2, fj =#fj syn2)
flat' (Leaf n) =
  \c nl (l=l, j=j) ->
      (fl= c n l, fj= c n j)

foldr :: (Integer→a→a) → a → L → a
foldr k z Nil = z
foldr k z (Cons(n,l)) = k n (foldr k z l)

build :: ((Integer→L→L) → L → a) → a
build g = g (Cons) Nil
```

where `(Cons)` denotes the curried list constructor function.

Then `revflat` are redefined as follows, by means of `foldr`/`build` operators.

```
revflat x =
  #rev ((let r = build (flat' x)
                        (l=#fj r,j=Nil)
         in foldr c nl (#fl r))
        (h=Nil))
  where c = \n l -> \(h=h) ->
              let syn = l (h= Cons(n,h))
              in (rev=#rev syn)
        nl = \(h=h) -> (rev=h)
```

The following rewrite rule of the shortcut fusion allows us to cancel `foldr`/`build` pairs.

$$\begin{aligned}
&\mathtt{foldr}\ k\ z\ (\#j_q\ ((\mathtt{build}\ g)(l_1 = e_1, \cdots, l_n = e_n))) \\
&= \#j_q\ ((g\ k\ z) \\
&\qquad (l_1 = \mathtt{foldr}\ k\ z\ e_1, \cdots, l_n = \mathtt{foldr}\ k\ z\ e_n))
\end{aligned}$$

Notice that this rewrite rule is different from that of the original one: Auxiliary record operations are involved, since input and output of the recursive function definition involves those operations. For the correctness of the altered transformation rule, see Appendix A.

```
data T = Node (T,T) | Leaf Integer
data L = Cons (Integer,L) | Nil

flat :: T → (l:: L, j :: L)
          → (fl:: L, fj:: L)
flat (Node(t1,t2)) (l=l, j=j) =
  let syn1 = flat t1 (l=l, j=j)
      syn2 = flat t2
               (l= #fl syn1, j= #fj syn1)
  in (fl= #fl syn2, fj= #fj syn2)
flat (Leaf n) (l=l, j=j) =
  (fl= Cons(n,l), fj= Cons(n,j))
```

```
rev :: L → (h:: L) → (rev:: L)
rev (Cons(n,l)) (h=h) =
  let syn = rev l (h= Cons(n,h))
  in (rev= #rev syn)
rev Nil (h=h) = (rev=h)

revflat :: T → L
revflat x =
  #rev (rev
         (let r= flat x (l= #fj r, j=Nil)
          in #fl r)
         (h=Nil))
```

Figure 1: The source program

The above rewrite rule solely is not enough for composing `fold` and `rev` functions, however. This is because, in the expression **`let`** `r= build ... in foldr c nl (#fl r)`, the `foldr`/`build` pair is syntactically separated by the **`let`** construct. Even worse this **`let`** definition is circular, and therefore naïve inlining of the **`let`** definition does not work (Inlining circular definition causes an infinite unfolding.)

We solve this problem by making explicit *hidden* `foldr`/`build` pairs using type information. In the present case, the expression `build` $\cdots$ bound by `r` has type `(fl :: L, fj :: L)`, which makes it possible to create a `foldr`/`build` pair for each record label `fl` and `fj`, as follows.

```
let r = build (flat' x) (l=#fj r,j=Nil)
    rfl = foldr c nl
            (#fl (build (flat' x)
                    (l=#fj r,j=Nil)))
    rfj = foldr c nl
            (#fj (build (flat' x)
                    (l=#fj r,j=Nil)))
in rfl
```

Here the new variables `r`*l* (where *l* is either `fl` or `fj`) denote the `foldr`/`build` pair for the label *l*.

Applying the shortcut fusion rule, we obtain the following program.

```
let r = build (flat' x) (l=#fj r,j=Nil)
    rfl = #fl (flat' x c nl
                (l= foldr c nl (#fj r),
                 j= foldr c nl Nil))
    rfj = #fj (flat' x c nl
                (l= foldr c nl (#fj r),
                 j= foldr c nl Nil))
in rfl
```

Since `foldr c nl (#fj r)` is equal to `rfj` and `r` is no more referenced, we have:

```
let rfl = #fl (flat' x c nl
                (l= rfj,
                 j= foldr c nl Nil))
    rfj = #fj (flat' x c nl
                (l= rfj,
                 j= foldr c nl Nil))
in rfl
```

As `rfl` and `rfj` have a common subexpression `(flat' ...)`, this can be further simplified:

```
let r = flat' x c nl
          (l=#fj r, j= foldr c nl Nil)
in #fl r
```

With additional simplifications, we obtain the following result of shortcut fusion written in the normal recursion style.

```
rf ::
  T → (j:: (h:: L) → (rev:: L),
       l:: (h:: L) → (rev:: L))
    → (fj:: (h:: L) → (rev:: L),
       fl:: (h:: L) → (rev:: L))
rf (Node(t1,t2)) inh =
  let syn1 = rf t1 (l=#l inh, j=#j inh)
      syn2 = rf t2 (l=#fl syn1, j=#fj syn1)
  in (fl=#fl syn2, fj=#fj syn2)
rf (Leaf n) inh =
  (fl= \(h=h) ->
         let syn = (#l inh) (h=Cons(n,h))
         in (rev=#rev syn),
   fj= \(h=h) ->
         let syn = (#j inh) (h=Cons(n,h))
         in (rev=#rev syn))

revflat :: T → L
revflat x =
  #rev ((let r = rf x
                  (l=#fj r,
```

```
                      j= \y->(rev=#h y))
         in #fl r)
        (h=Nil))
```

We remark that deforestation by the shortcut fusion improves nothing: It only replaced intermediate data by function closures. The subsequent transformation steps, which are our contribution, are devoted to remove the function closures.

2.3 Deforesting in accumulating parameters by function closure elimination

The fact that the result of the preceding transformation introduces function closure constructions is reflected by the type of function `rf`:

$$(\mathtt{l} :: (\mathtt{h} :: \mathtt{L}) \to (\mathtt{rev} :: \mathtt{L}), \mathtt{j} :: (\mathtt{h} :: \mathtt{L}) \to (\mathtt{rev} :: \mathtt{L})) \;\; \to$$
$$(\mathtt{fl} :: (\mathtt{h} :: \mathtt{L}) \to (\mathtt{rev} :: \mathtt{L}), \mathtt{fj} :: (\mathtt{h} :: \mathtt{L}) \to (\mathtt{rev} :: \mathtt{L})),$$

which involves second-order function types.

The goal of the rest of our transformation process is to remove higher-order function types. When the transformation completes, we expect that the function `rf` has a first-order type:

$$(\mathtt{flh} :: \mathtt{l}, \mathtt{fjh} :: \mathtt{L}, \mathtt{lrev} :: \mathtt{L}, \mathtt{jrev} :: \mathtt{L}) \;\; \to$$
$$(\mathtt{flrev} :: \mathtt{L}, \mathtt{fjrev} :: \mathtt{L}, \mathtt{lh} :: \mathtt{L}, \mathtt{jh} :: \mathtt{L}),$$

where `flh`, `fjh`, etc. are new field labels generated by taking a "product" of record labels, where a label l from the function `flat` and j from `rev` together give a new label $l\,j$.

This transformation is performed in three substeps: *function type lifting*, *type flattening*, and *higher-order removal*. In the following, we will show each transformation step in turn, where a special attention should be paid to types, which suggest the appropriate direction of transformation.

2.3.1 Function type lifting

The first subtransformation is *function type lifting*, which *lifts* labels of inner record types outside of function closures and distributes the labels of outer record types on each side of function arrow. This is intended to make the outer field labels and the inner ones interact each other in the next step of transformation. In the present example, the type

$$(\mathtt{l} :: (\mathtt{h} :: \mathtt{L}) \to (\mathtt{rev} :: \mathtt{L}), \mathtt{j} :: (\mathtt{h} :: \mathtt{L}) \to (\mathtt{rev} :: \mathtt{L})) \;\; \to$$
$$(\mathtt{fl} :: (\mathtt{h} :: \mathtt{L}) \to (\mathtt{rev} :: \mathtt{L}), \mathtt{fj} :: (\mathtt{h} :: \mathtt{L}) \to (\mathtt{rev} :: \mathtt{L}))$$

is conerted into

$$(\langle \mathtt{l} :: (\mathtt{h} :: \mathtt{L}),\;\; \mathtt{j} :: (\mathtt{h} :: \mathtt{L})\rangle \;\; \to$$
$$\quad \langle \mathtt{l} :: (\mathtt{rev} :: \mathtt{L}),\;\; \mathtt{j} :: (\mathtt{rev} :: \mathtt{L})\rangle) \;\; \to$$
$$(\langle \mathtt{l} :: (\mathtt{h} :: \mathtt{L}),\;\; \mathtt{j} :: (\mathtt{h} :: \mathtt{L})\rangle \;\; \to$$
$$\quad \langle \mathtt{fl} :: (\mathtt{rev} :: \mathtt{L}),\;\; \mathtt{fj} :: (\mathtt{rev} :: \mathtt{L})\rangle).$$

The function type lifting transformation is accomplished by the following set of rewrite rules. (We write $\mathrm{out}l\, M$ for **case** M **of** $\langle l = y\rangle \to y$, where $\mathrm{out}l\, M$ is intended to drop the injection label l of M.)

$$\begin{array}{rcl}
(\mathtt{fl}{=}M_1, \mathtt{fj}{=}M_2) & \Longrightarrow & \backslash \mathtt{x} {\to} \mathbf{case}\; x \;\mathbf{of} \\
 & & \quad \langle \mathtt{fl} = y\rangle \to \langle \mathtt{fl} = M_1\; y\rangle \\
 & & \quad \langle \mathtt{fj} = y\rangle \to \langle \mathtt{fj} = M_2\; y\rangle \\
(\mathtt{l}{=}M_1, \mathtt{j}{=}M_2) & \Longrightarrow & \backslash \mathtt{x} {\to} \mathbf{case}\; x \;\mathbf{of} \\
 & & \quad \langle \mathtt{l} = y\rangle \to \langle \mathtt{l} = M_1\; y\rangle \\
 & & \quad \langle \mathtt{j} = y\rangle \to \langle \mathtt{j} = M_2\; y\rangle \\
\#l\;\; M & \Longrightarrow & \backslash \mathtt{x} {\to} \mathrm{out}l\,(M\;\; \langle l = \mathtt{x}\rangle) \\
 & & (l \text{ is either } \mathtt{fl}, \mathtt{fj}, \mathtt{l}, \text{ or } \mathtt{j})
\end{array}$$

The rewrite rules are applied to inner expressions first.

These rewrite rules are informally justified by the following type-based reasoning. The first rule is intended to transform a record expression into a function that encodes the functionality of record field selection. The encoding function expects a variant value $\langle l = v\rangle$ as input, where the injection label l is the field to select and the injected value v is the argument passed to the selected function (The selected value is guaranteed to be a function by the type of record $(\mathtt{fl} :: (\mathtt{h} :: \mathtt{L}) \to (\mathtt{rev} :: \mathtt{L})), \mathtt{fj} :: (\mathtt{h} :: \mathtt{L}) \to (\mathtt{rev} :: \mathtt{L})))$). The result is returned as a variant value injected by the same label l of the input. The second rule is justified in the same way. For the third rule, the record expression M, after transformation, is expected to be a function that encodes field selection in the way described above. Therefore we trigger the field selection functionality of the encoding function by passing a variant injected by the label to select. The returned result comes with an injection label, which is removed by the out$l$ operation.

Applying this transformation, we obtain the following program.

```
rf::
 T-> (⟨l:: (h:: L), j :: (h:: L)⟩
       → ⟨l:: (rev:: L), j:: (rev:: L)⟩)
 → (⟨fl:: (h:: L), fj:: (h:: L)⟩
       → ⟨fl:: (rev:: L), fj:: (rev:: L)⟩)
rf (Node(t1,t2)) inh =
  let syn1 = rf t1
        (\x -> case x of
            ⟨l=y⟩ -> ⟨l=outl (inh ⟨l=y⟩)⟩
            ⟨j=y⟩ -> ⟨j=outj (inh ⟨j=y⟩)⟩)
```

```
      syn2 = rf t2
       (\x -> case x of
          ⟨l=y⟩ -> ⟨l=outfl (syn1 ⟨fl=y⟩)⟩
          ⟨j=y⟩ -> ⟨j=outfj (syn1 ⟨fj=y⟩)⟩)
  in \x -> case x of
        ⟨fl=y⟩ -> ⟨fl=outfl (syn2 ⟨fl=y⟩)⟩
        ⟨fj=y⟩ -> ⟨fj=outfj (syn2 ⟨fj=y⟩)⟩)
rf (Leaf n) inh =
  \x -> case x of
    ⟨fl=y⟩ ->
      let syn = outl (inh
                  ⟨l=(h=Cons(n, #h y))⟩)
      in ⟨fl=(rev=#rev syn)⟩
    ⟨fj=y⟩ ->
      let syn = outj (inh
                  ⟨j=(h=Cons(n, #h y))⟩)
      in ⟨fj=(rev=#rev syn)⟩

revflat x =
  #rev ((let r = rf x
          (\x -> case x of
             ⟨l=y⟩ -> ⟨l=outfj (r ⟨fj=y⟩)⟩
             ⟨j=y⟩ -> ⟨j=(rev=#h y)⟩)
        in \z -> outfl (r ⟨fl=z⟩))
        (h=Nil))
```

2.3.2 Type flattening

The next step of transformation is *type flattening*. This transformation is intended to replace every nested record/variant constructor pair with a single record constructor, and also to replace every nested record/variant destructor pair with a single record field selector.

At the type level, this corresponds to conversion from

$$\langle l_1 :: (j_1 :: \tau, \cdots, j_m :: \tau), \cdots, l_n :: (j_1 :: \tau, \cdots, j_m :: \tau)\rangle$$

into

$$(l_1 j_1 :: \tau, \cdots, l_1 j_m :: \tau, \cdots, l_n j_1 :: \tau, \cdots, l_n j_m :: \tau),$$

where the result is a record type whose set of labels are constructed by an elementwise product. In the present case, the type

$$\begin{array}{l} (\langle \mathtt{l} :: (\mathtt{h} :: \mathtt{L}),\ \ \mathtt{j} :: (\mathtt{h} :: \mathtt{L})\rangle \ \to \\ \quad \langle \mathtt{l} :: (\mathtt{rev} :: \mathtt{L}),\ \ \mathtt{j} :: (\mathtt{rev} :: \mathtt{L})\rangle) \ \to \\ (\langle \mathtt{l} :: (\mathtt{h} :: \mathtt{L}),\ \ \mathtt{j} :: (\mathtt{h} :: \mathtt{L})\rangle \ \to \\ \quad \langle \mathtt{fl} :: (\mathtt{rev} :: \mathtt{L}),\ \ \mathtt{fj} :: (\mathtt{rev} :: \mathtt{L})\rangle) \end{array}$$

is converted to

$$\begin{array}{l} ((\mathtt{lh} :: \mathtt{L},\ \ \mathtt{jh} :: \mathtt{L}) \to (\mathtt{lrev} :: \mathtt{L}, \mathtt{jrev} :: \mathtt{L})) \ \to \\ ((\mathtt{flh} :: \mathtt{L},\ \ \mathtt{fjh} :: \mathtt{L}) \to (\mathtt{flrev} :: \mathtt{L}, \mathtt{fjrev} :: \mathtt{L})). \end{array}$$

To achieve this desired transformation, we rewrite every variant/record constructor pair by

$$\langle l = (j = M)\rangle \implies (lj = M),$$

and every variant/record destructor pair by

$$\#j\ (\mathrm{out}l\ M) \implies \#lj\ M.$$

After canceling all record/variant pairs, we apply the following rewrite rules at last:

$$\begin{array}{l} \backslash x \to \textbf{case}\ x\ \textbf{of}\ \ \langle \mathtt{fl} = y\rangle \to M_1 \\ \qquad\qquad\qquad\quad \langle \mathtt{fj} = y\rangle \to M_2 \qquad \implies \\ \quad \backslash y \to (\mathtt{flrev=\#flrev}\ \ M_1[y/\mathtt{(h=\#flh\ y)}], \\ \qquad\quad \mathtt{fjrev=\#fjrev}\ \ M_2[y/\mathtt{(h=\#fjh\ y)}]) \end{array}$$

$$\begin{array}{l} \backslash x \to \textbf{case}\ x\ \textbf{of}\ \ \langle \mathtt{l} = y\rangle \to M_1 \\ \qquad\qquad\qquad\quad \langle \mathtt{j} = y\rangle \to M_2 \qquad \implies \\ \quad \backslash y \to (\mathtt{lrev=\#lrev}\ \ M_1[y/\mathtt{(h=\#lh\ y)}], \\ \qquad\quad \mathtt{jrev=\#jrev}\ \ M_2[y/\mathtt{(h=\#jh\ y)}]) \end{array}$$

where $M[y/N]$ is an expression obtained by replacing every free variable y in M by N.

Here we need to remark two points. First, the rewrite rule for variant/record constructor pair produces a record with an incomplete set of labels. For example, an expression $\langle \mathtt{l} = (\mathtt{h} = M)\rangle$ rewrites to a record $(\mathtt{lh} = M)$, which has a missing complementary label `lrev`. We insist that a record construction with missing fields is well-typed by assuming that missing fields are implicitly defined to have a diverging expression Ω of appropriate type τ.

Second, the above rewrite rules stick to a syntactic property that constructor/destructor pairs are placed side by side, which is not always the case. However, we can refine the above rules so that they do not rely on such syntactic property, by using the extensionality of records:

$$\begin{array}{l} \text{if } M \text{ has type } (l_1 :: \tau_1, \ldots, l_n :: \tau_n) \text{ then} \\ M = (l_1 = \#l_1\ M, \cdots, l_n = \#l_n\ M). \end{array}$$

The refined rewrite rules are:

$$\begin{array}{l} \text{if } M \text{ has type } (j_1 :: \tau_1, \ldots, j_n :: \tau_n) \text{ then} \\ \ \ \langle l = M\rangle \implies (lj_1 = \#j_1 M, \ldots, lj_n = \#j_n M) \\ \ \ \mathrm{out}l\ M \quad \implies (j_1 = \#lj_1 M. \ldots, j_n = \#lj_n M). \end{array}$$

Note that the former set of rules are special cases of this refined set of rules.

The current transformation may introduce redundant record constructors and destructors, but they can be eliminated by using the extensionality of records. The only difficult case is that recursive **let** definition is involved. In this case, however, one can follow a similar strategy taken in Section 2.2 to put the desired pairs side by side.

The validity of the above rewrite rules can be ensured informally by a type-based reasoning. The aim of transformation is to coerce every type of the form $\langle l_1 :: (h_1 :: \tau, \ldots), l_2 :: (h_1 :: \tau, \ldots), \cdots \rangle$ into $(l_1 h_1 :: \tau, \ldots, l_i h_j :: \tau, \ldots)$, and hence a nested record/variant constructor pair (destructor pair, *resp.*) should be translated into a single record constructor (destructor, *resp.*) with concatenated field labels. For case expressions, each of them is expected, after transformation, to accept records of type $(l_1 h_1 :: \tau, \ldots, l_i h_j :: \tau, \ldots)$. However, every expression in a case branch $\langle l = y \rangle \Rightarrow \cdots$ expects to receive a record of type $(h_1 :: \tau, \ldots)$ through y. To fix this mismatch, we replace every occurrence of y by a record with relevant fields, *i. e.*, $(h_1 = \#lh_1\,\mathtt{y}, \ldots, h_i = \#lh_i\,\mathtt{y}, \ldots)$.

The overall result of the type flattening transformation is given below.

```
rf ::
  T → ((lh:: L, jh :: L)
           → (jrev:: L, lrev :: L))
    → ((fjh:: L, flh :: L)
           → (fjrev:: L, flrev :: L))
rf (Node(t1,t2)) = \inh ->
  let syn1 = rf t1
       (\y ->
        (lrev=#lrev (inh (lh=#lh y)),
         jrev=#jrev (inh (jh=#jh y))))
      syn2 = rf t2
       (\y ->
        (lrev=#flrev (syn1 (flh=#lh y)),
         jrev=#fjrev (syn1 (fjh=#jh y))))
  in \y ->
      (flrev=#flrev (syn2 (flh=#flh y)),
       fjrev=#fjrev (syn2 (fjh=#fjh y)))
rf (Leaf n) = \inh ->
  \y -> (flrev=#lrev (inh
                 (lh=(Cons(n,#flh y)))),
         fjrev=#jrev (inh
                 (jh=(Cons(n,#fjh y)))))

revflat x =
  let r = rf x
        (\y ->
          (lrev=#fjrev (r (fjh=(#lh y))),
           jrev=#jh y))
  in #flrev (r (flh=(#h (h=Nil))))
```

Here some redundant record constructor/destructor pairs are eliminated using the record extensionality. Further the second argument inh of the recursive function rf is curried for the convenience of the next transformation.

2.3.3 Higher-order removal

Now we are ready to eliminate higher-order functions. This final transformation step is intended to convert higher-order function type

$$((\mathtt{lh} :: \mathtt{L},\ \ \mathtt{jh} :: \mathtt{L}) \rightarrow (\mathtt{lrev} :: \mathtt{L}, \mathtt{jrev} :: \mathtt{L})) \ \rightarrow$$
$$((\mathtt{flh} :: \mathtt{L},\ \ \mathtt{fjh} :: \mathtt{L}) \rightarrow (\mathtt{flrev} :: \mathtt{L}, \mathtt{fjrev} :: \mathtt{L}))$$

into

$$(\mathtt{fjh} :: \mathtt{L}, \mathtt{flh} :: \mathtt{L}, \mathtt{jrev} :: \mathtt{L}, \mathtt{lrev} :: \mathtt{L}) \ \rightarrow$$
$$(\mathtt{fjrev} :: \mathtt{L}, \mathtt{flrev} :: \mathtt{L}, \mathtt{lh} :: \mathtt{L}, \mathtt{jh} :: \mathtt{L}),$$

a first-order function type from record to record. Notice that the labels occurring negatively in the original type constitutes the argument record and that those occurring positively does the result record.

The transformation process crucially relies on a particular syntactic property that the recursive function rf of type $\mathtt{T} \rightarrow ((\mathtt{lh} :: \mathtt{L}, \mathtt{jh} :: \mathtt{L}) \rightarrow (\mathtt{lrev} :: \mathtt{L}, \mathtt{jrev} :: \mathtt{L})) \rightarrow ((\mathtt{flh} :: \mathtt{L}, \mathtt{fjh} :: \mathtt{L}) \rightarrow (\mathtt{flrev} :: \mathtt{L}, \mathtt{fjrev} :: \mathtt{L}))$ is defined for every constructor of T by the following syntactic form:

$$\begin{array}{l} \backslash \mathtt{x}_0 \ \mathtt{->}\ \mathbf{let}\ \mathtt{x}_1\mathtt{=}\ \mathtt{rf}\ N_1\ \ (\backslash \mathtt{y}_1\mathtt{->}M_1) \\ \qquad\qquad\qquad \vdots \\ \qquad\qquad\quad \mathtt{x}_n\mathtt{=}\ \mathtt{rf}\ N_n\ \ (\backslash \mathtt{y}_n\mathtt{->}M_n) \\ \qquad\quad \mathbf{in}\ \backslash \mathtt{y}_0\mathtt{->}M_0 \end{array} \qquad (*)$$

where the variables $\mathtt{y}_i$'s and $\mathtt{x}_i$'s are distinct each other, and expressions $M_0, \ldots, M_n$ must meet the following additional conditions:

- They contain no λ-abstraction or variant function, and
- Each expression M_i has no free occurrence of variables other than $\mathtt{x}_0, \ldots, \mathtt{x}_n$, and $\mathtt{y}_i$.

Every syntactic unit that matches the syntactic form $(*)$ is then transformed as follows.

1. For every $\mathtt{x}_i$ $(i \neq 0)$ and every label l of either flh or fjh, identify a unique subexpression of the form $\mathtt{x}_i\ (l = M_{i,l})$ (in expressions $M_0, \ldots, M_n$).
2. For every label l of either lh or jh, identify a unique subexpression of the form $\mathtt{x}_0\ (l = M_l)$ (in expressions $M_0, \ldots, M_n$).

3. Finally, rewrite the whole expression $(*)$ as follows:

```
\x_0 ->
 let x_1= rf N_1
       (lrev=#lrev $\overline{M_1}$, jrev=#jrev $\overline{M_1}$,
        flh=$\overline{M_{1,\mathtt{flh}}}$, flj=$\overline{M_{1,\mathtt{flj}}}$)
                  ⋮
      x_n= rf N_n
       (lrev=#lrev $\overline{M_n}$, jrev=#jrev $\overline{M_n}$,
        flh=$\overline{M_{n,\mathtt{flh}}}$, flj=$\overline{M_{n,\mathtt{flj}}}$)
 in (flrev=#flrev $\overline{M_0}$, fjrev=#fjrev $\overline{M_0}$,
      lh=$\overline{M_{\mathtt{lh}}}$, jh=$\overline{M_{\mathtt{jh}}}$)
```

where $\overline{M}$ is the result of applying the following rewrite rules:

$$\begin{array}{lll} \mathtt{x}_i\, M & \Longrightarrow \mathtt{x}_i & \text{(for every } i\text{)} \\ \mathtt{y}_i & \Longrightarrow \mathtt{x}_i & \text{(for every } i\text{)} \end{array}$$

The idea behind these rewrite rules is as follows. What we want to do is to convert every higher-order function type $(\alpha \to \beta) \to (\gamma \to \delta)$ into $\beta@\gamma \to \alpha@\delta$, where $\alpha, \beta, \gamma, \delta$ are record types and @ denotes concatenation of them. For this conversion, we "steal" arguments applied to functions of type either $\alpha \to \beta$ or $\gamma \to \delta$. (The rewrite steps 2 and 3 identify such application sites.) To convert the type $\gamma \to \delta$ of variable $\mathtt{x}_i$ $(i \neq 0)$ to $\alpha@\delta$, we rewrite the second argument of $\mathtt{x}_i$'s defining expression (*i. e.*, the λ-expression $\backslash\mathtt{y}_i\texttt{->}M_i$) into a record expression, whose field expressions for the record labels of β are drawn from M_i and for those of γ from the "stolen" expressions. We similarly transform the expression $\backslash\mathtt{y}_0\texttt{->}M_0$ into a record expression, whose field expressions for the record labels of δ are drawn from M_0 and for those of α from the "stolen" expressions. Every application site $\mathtt{x}_i\, M$, since the argument M is stolen, is rewrote $\mathtt{x}_i$. Every reference to variable $\mathtt{y}_i$ is replaced by $\mathtt{x}_i$ (of type $\alpha@\delta$ for $i = 0$; $\gamma@\beta$ otherwise), since the denotation of $\mathtt{x}_i$ (of type δ for $i = 0$; β otherwise) is unified with that of $\mathtt{y}_i$ (of type α for $i = 0$; γ otherwise) by the record concatenation.

Notice that the definition of function `revflat` contains a call of `rf`, which does not match the syntactic pattern $(*)$. The function body of `revflat` is transformed as follows. Suppose every function call of `rf` outside the recursive definition of `rf` has a syntactic form:

```
let x_1= rf N_1 (\y_1->M_1)
            ⋮                         —— (*')
      x_n= rf N_n (\y_n->M_n)
   in M_0
```

where the same conditions as $(*)$ are imposed (except for that in the rewrite rule 2). Then, this is rewrote to the following expression.

```
let x_1= rf N_1
      (lrev=#lrev $\overline{M_1}$, jrev=#jrev $\overline{M_1}$,
       flh=$\overline{M_{1,\mathtt{flh}}}$, flj=$\overline{M_{1,\mathtt{flj}}}$)
                 ⋮
     x_n= rf N_n
      (lrev=#lrev $\overline{M_n}$, jrev=#jrev $\overline{M_n}$,
       flh=$\overline{M_{n,\mathtt{flh}}}$, flj=$\overline{M_{n,\mathtt{flj}}}$)
in $\overline{M_0}$
```

Applying the rewrite rules, we obtain the final result:

```
rf ::
 T → (fjh:: L, flh :: L,
       jrev :: L, lrev :: L)
   → (flrev:: L, fjrev :: L,
       lh :: L, jh :: L)
rf (Node(t1,t2)) inh =
  let syn1 = rf t1
         (lrev=#lrev inh, jrev=#jrev inh,
          flh=#lh syn2, fjh=#jh syn2)
      syn2 = rf t2
         (lrev=#flrev syn1,
          jrev=#fjrev syn1,
          flh=#flh inh, fjh=#fjh inh)
  in (flrev=#flrev syn2,
      fjrev=#fjrev syn2,
      lh=#lh syn1, jh=#jh syn1)
rf (Leaf n) inh =
  (flrev=#lrev inh, fjrev=#jrev inh,
   lh=Cons(n,#flh inh),
   jh=Cons(n,#fjh inh))

revflat x =
  let r = rf x
           (lrev=#fjrev r, jrev=#jh r,
            flh=Nil, fjh=#lh r)
  in #flrev r
```

3 Formal Definition of Transformation

This section formally defines the transformations in the previous section for a call-by-name language.

In the following, a vector notation $\overrightarrow{a}$ is used to represent a (possibly null) sequence of syntactic objects

$a_1,\ldots,a_n$. We allow to confuse a sequence with a set, say, $a \in \overrightarrow{a}$ to mean $a \in \{a_1,\ldots,a_n\}$. Further, a two vector notations juxtaposed together represents a sequence of pairs of syntactic objects, say, $(\overrightarrow{l} = \overrightarrow{M})$ to mean $(l_1 = M_1,\ldots,l_n = M_n)$.

3.1 Types

The set of type expressions are defined by the following grammar.

$$
\begin{array}{rlll}
\tau & ::= & \alpha & \text{type variable} \\
 & \mid & \tau \to \tau & \text{function type} \\
 & \mid & (\overrightarrow{l} : \overrightarrow{\tau}) & \text{record type} \\
 & \mid & \langle l : \tau \rangle & \text{variant type} \\
 & \mid & \langle \overrightarrow{l} : \overrightarrow{\tau} \rangle \Rightarrow \langle \overrightarrow{l} : \overrightarrow{\tau'} \rangle & \text{variant function type} \\
 & \mid & \nu\alpha.\langle \mathtt{C}_1(\tau_1),\ldots,\mathtt{C}_n(\tau_n)\rangle & \text{inductive type}
\end{array}
$$

The types are monomorphic ones only and they are quite standard, except for the last three of them. We allow a variant type to have only a single injection label. Variants are processed by a function that has a variant function type, which represents a special class of injection-preserving functions, *i. e.*, functions that map a value of type $\langle l = \tau\rangle$ into $\langle l = \tau'\rangle$. An inductive type defines an algebraic type with constructors $\mathtt{C}_1,\ldots,\mathtt{C}_n (n \geq 1)$. For example, the type of list, which is defined in Haskell notation `data L = Nil | Cons τ * L`, is expressed by $\nu\alpha.\langle \mathtt{Nil}(), \mathtt{Cons}(\mathtt{head} = \tau, \mathtt{tail} = \alpha)\rangle$, where the record type $(\mathtt{head} = \tau, \mathtt{tail} = \alpha)$ represents a cons cell. We consider only covariant inductive types (*i. e.*, the type variable bound by ν occurs only in positive positions) and also assume that no different inductive types share the same constructor names.

A type variable α occurring in a type τ is said to be *free*, if it is not bound by any $\nu\alpha.\cdots$. A type is said to be *closed*, if it has no free occurrences of type variables. Though we are concerned with closed types only, we need type substitution to unroll inductive types: A *type substitution* of τ' for every free occurrence of type variable α in τ, written $\tau[\alpha/\tau']$, is defined as usual by induction on the structure of τ.

3.2 The language expressions

The set of expressions of the language is defined below.

$$
\begin{array}{rlll}
M & ::= & x & \text{variable} \\
 & \mid & \lambda x : \tau.M & \text{abstraction} \\
 & \mid & MM & \text{application} \\
 & \mid & \mathbf{fix}\, M & \text{fixpoint recursion} \\
 & \mid & (\overrightarrow{l} = \overrightarrow{M}) & \text{record construction} \\
 & \mid & \#l\ M & \text{field selection} \\
 & \mid & \langle l = M \rangle & \text{variant construction} \\
 & \mid & \mathbf{fun}\overrightarrow{\langle l = x : \tau\rangle} \Rightarrow \overrightarrow{M} & \text{variant function} \\
 & \mid & \mathrm{out}l\ M & \text{variant destruction} \\
 & \mid & \mathtt{C}(M) & \text{datatype construction} \\
 & \mid & \mathbf{case}\, x\, \mathbf{of} \overrightarrow{\mathtt{C}(x)} \Rightarrow \overrightarrow{M} & \text{datatype destruction}
\end{array}
$$

The language is an explicitly typed lambda-calculus (where all annotated types are assumed to be closed), extended with records, variants, fixpoint, and datatype operators.

A variable occurrence of x in M is said to be *free*, if x is not bound any λ- or **fun**-bindings. An expression with no free variables is called *closed*. We write $M[x/N]$ for capture-free substitution of N for free occurrences of x in N.

Most of the language constructs are quite standard, except for variant destruction and variant function: A variant destruction is just an inverse of variant construction, *i. e.*, $\mathrm{out}l\langle l = M\rangle = M$. A variant function

$$\mathbf{fun}\,\langle l_1 = x_1\rangle \Rightarrow M_1;\ldots;\langle l_n = x_n\rangle \Rightarrow M_n$$

roughly corresponds to the following Haskell expression

$$\texttt{\textbackslash x -> } \mathbf{case}\, \mathtt{x}\, \mathbf{of}\ \langle l_1 = x_1\rangle \to M_1;\ldots;\langle l_n = x_n\rangle \to M_n$$

with a constraint that each M_i has type $\langle l_i : \tau_i\rangle$. This constraint is posed by a variant function type, which cannot be expressed by the other type constructs.

The language is missing the feature of mutually recursive circular **`let`** definition, but it can be expressed in the present language. That is,

$$\mathbf{let}\ x_1 = M_1;\ \ldots;\ x_n = M_n\ \mathbf{in}\ M$$

can be encoded using fixpoint:

$$\mathbf{fix}\,(\lambda r : \tau.(\overline{x_1} = M_1[x_1/\#\overline{x_1}\ r],\ldots,\overline{x_n} = M_n[x_n/\#\overline{x_n}\ r]))$$

where $\overline{x_i}'s$ are field labels and τ is an appropriate record type.

$$\frac{\Gamma(x)=\tau}{\Gamma\vdash x:\tau} \qquad \frac{\Gamma\cup x:\tau\vdash M:\tau'}{\Gamma\vdash \lambda x:\tau.M:\tau\to\tau'} \qquad \frac{\Gamma\vdash M:\tau'\to\tau \quad \Gamma\vdash N:\tau'}{\Gamma\vdash MN:\tau} \qquad \frac{\Gamma\vdash M:\tau\to\tau}{\Gamma\vdash \mathbf{fix}\,M:\tau}$$

$$\frac{\Gamma\vdash M_i:\tau_i \text{ (for every } i)}{\Gamma\vdash(\overrightarrow{l}=\overrightarrow{M}):(\overrightarrow{l}:\overrightarrow{\tau})} \qquad \frac{\Gamma\vdash M:(l_1:\tau_1,\ldots,l_n:\tau_n)}{\Gamma\vdash \#l_i\,M:\tau_i}$$

$$\frac{\Gamma\vdash M:\tau}{\Gamma\vdash\langle l=M\rangle:\langle l:\tau\rangle} \qquad \frac{\Gamma\cup x_i:\tau_i\vdash M:\langle l_i:\tau_i'\rangle \text{ (for every } i)}{\Gamma\vdash \mathbf{fun}\overrightarrow{\langle l=x:\tau\rangle}\Rightarrow\overrightarrow{M}:\langle\overrightarrow{l}:\overrightarrow{\tau}\rangle\Rightarrow\langle\overrightarrow{l}:\overrightarrow{\tau'}\rangle}$$

$$\frac{\Gamma\vdash M:\langle\overrightarrow{l}:\overrightarrow{\tau}\rangle\Rightarrow\langle\overrightarrow{l}:\overrightarrow{\tau'}\rangle \quad \Gamma\vdash N:\langle l_i:\tau_i\rangle}{\Gamma\vdash MN:\langle l_i:\tau_i'\rangle} \qquad \frac{\Gamma\vdash M:\langle l:\tau\rangle}{\Gamma\vdash \mathrm{out}l\,M:\tau}$$

Let $\theta=[\alpha/\nu\alpha.\langle\mathtt{C}_1(\tau_1),\ldots,\mathtt{C}_n(\tau_n)\rangle]$, then

$$\frac{\Gamma\vdash M:\tau_i\theta}{\Gamma\vdash\mathtt{C}_i(M):\nu\alpha.\langle\mathtt{C}_1(\tau_1),\ldots,\mathtt{C}_n(\tau_n)\rangle} \qquad \frac{\begin{array}{c}\Gamma(x)=\nu\alpha.\langle\mathtt{C}_1(\tau_1),\ldots,\mathtt{C}_n(\tau_n)\rangle\\ \Gamma\cup x_i:\tau_i\theta\vdash M_i:\tau \text{ (for every } i)\end{array}}{\Gamma\vdash\mathbf{case}\,x\,\mathbf{of}\,\mathtt{C}(x_1)\Rightarrow M_1;\ldots;\mathtt{C}(x_n)\Rightarrow M_n:\tau}$$

Figure 2: Typing Rules

$$\frac{}{V\Downarrow V} \qquad \frac{M\Downarrow\lambda x:\tau.F \quad F[N/x]\Downarrow V}{MN\Downarrow V} \qquad \frac{M(\mathbf{fix}\,M)\Downarrow V}{\mathbf{fix}\,M\Downarrow V}$$

$$\frac{M\Downarrow(\vec{l}=\vec{M}) \quad M_i\Downarrow V}{\#l_i\,M\Downarrow V} \qquad \frac{M\Downarrow\mathbf{fun}\overrightarrow{\langle l=x:\tau\rangle}\Rightarrow\overrightarrow{F} \quad N\Downarrow\langle l_i=L\rangle \quad F_i[L/x]\Downarrow V}{MN\Downarrow V}$$

$$\frac{M\Downarrow\langle l=N\rangle}{\mathrm{out}l\,M\Downarrow N} \qquad \frac{M\Downarrow\mathtt{C}_i(N) \quad M_i[N/x_i]\Downarrow V}{\mathbf{case}\,M\,\mathbf{of}\,\mathtt{C}(x_1)\Rightarrow M_i;\ldots;\mathtt{C}(x_n)\Rightarrow M_n\Downarrow V}$$

Figure 3: Operational Semantics

3.3 Typing rules and operational semantics

Figure 2 gives the set of typing rules that defines the type assignment relation of the language. A *typing judgment* is a ternary relation $\Gamma\vdash M:\tau$, where Γ is a *type environment*, a finite map from variables to types, M is an expression, and τ is the type assigned to M. We write $\Gamma\cup\Gamma'$ for an extension of Γ by Γ'.

We give a call-by-name semantics for the language. The *evaluation relation* is expressed by a binary relation $M\Downarrow V$ where M and V are closed expression and V is a *value*, which belongs to the set of expressions defined below:

$$\begin{array}{rl} V ::= & \lambda x:\tau.M \mid (\overrightarrow{l}=\overrightarrow{M}) \mid \langle l=M\rangle \\ & \mid \mathbf{fun}\overrightarrow{\langle l=x:\tau\rangle}\Rightarrow\overrightarrow{M} \mid \mathtt{C}(\overrightarrow{M}). \end{array}$$

The evaluation relation is inductively defined on the structure of M as in Figure 3.

3.4 Definition of transformations

We formally define three transformations discussed in Section 2.3. Here we only mention the type correctness of transformations, leaving semantic correctness to intuitions explained in Section 2.3.

We suppose that the transformation source is a result of shortcut fusion in Section 2.2, which is applied to a compositions of two functions of type $\tau\to(\overrightarrow{h}:\sigma)\to(\overrightarrow{l}:\sigma)$ and $\sigma\to(\overrightarrow{k}:\delta)\to(\overrightarrow{j}:\delta)$, where τ, σ, and δ are inductive types and $\overrightarrow{h}$, $\overrightarrow{l}$, $\overrightarrow{k}$, $\overrightarrow{j}$ are disjoint set of labels. Then we can assume that the result of shortcut fusion is a program that contains a recursive function of type $\tau\to(\overrightarrow{h}:(\overrightarrow{k}:\delta)\to(\overrightarrow{j}:\delta))\to(\overrightarrow{l}:(\overrightarrow{k}:\delta)\to(\overrightarrow{j}:\delta))$.

Without loss of generality, we can assume any set of labels that constitutes a record type or a variant function type is uniquely determined by a label that belongs to the set. That is, for any sets of labels $\overrightarrow{l}$ and $\overrightarrow{j}$, $\overrightarrow{l}\cap\overrightarrow{j}\neq\emptyset$ implies $\overrightarrow{l}=\overrightarrow{j}$.

$$\mathcal{W}_1\llbracket \#i\,M \rrbracket = \lambda x : (\overrightarrow{k} : \mathcal{T}_1\llbracket\delta\rrbracket).\mathrm{out}i(\mathcal{W}_1\llbracket M\rrbracket\,\langle i = x\rangle)$$
$$\text{where } i \in \overrightarrow{h} \text{ or } i \in \overrightarrow{l}\,;\ x \text{ is fresh}$$

$$\begin{aligned} \mathcal{W}_1\llbracket (i_1 = M_1, \ldots, i_n = M_n) \rrbracket = \\ \mathbf{fun}\ \langle i_1 = y\rangle \to \langle i_1 = \mathcal{W}_1\llbracket M_1\rrbracket y\rangle; \\ \vdots \\ \langle i_n = y\rangle \to \langle i_n = \mathcal{W}_1\llbracket M_n\rrbracket y\rangle \\ \text{where } \overrightarrow{i} = \overrightarrow{h} \text{ or } \overrightarrow{i} = \overrightarrow{l}\,;\ y \text{ is fresh} \end{aligned}$$

$$\begin{aligned} \mathcal{T}_1\llbracket (\overrightarrow{i} : (\overrightarrow{k} : \delta) \to (\overrightarrow{j} : \delta)) \rrbracket = \\ \langle \overrightarrow{i} : (\overrightarrow{k} : \mathcal{T}_1\llbracket\delta\rrbracket)\rangle \Rightarrow \langle \overrightarrow{j} : (\overrightarrow{k} : \mathcal{T}_1\llbracket\delta\rrbracket)\rangle \\ \text{where } \overrightarrow{i} = \overrightarrow{h} \text{ or } \overrightarrow{i} = \overrightarrow{l} \end{aligned}$$

Figure 4: Function type lifting transformation

We may occasionally annotate expressions with their type assignment, written M^{τ}, to indicate certain transformation rule is dependent on type information.

3.4.1 Function type lifting transformation

Figure 4 gives the formal definition of the function type lifting transformation $\mathcal{W}_1$, with a corresponding type transformation function $\mathcal{T}_1$. The transformation functions are defined inductively defined on the structure of expressions and types, respectively. We exhibited significant induction cases only; The other cases simply recurse on the immediate substructures, keeping the outermost construct with possible adjustment on type annotation. The type transformation function $\mathcal{T}_1$ is generalized to type environments by $\mathcal{T}_1\llbracket x_1 : \tau_1, \ldots, x_n : \tau_n\rrbracket = x_1 : \mathcal{T}_1\llbracket\tau_1\rrbracket, \ldots, x_n : \mathcal{T}_1\llbracket\tau_n\rrbracket$.

The type correctness of this transformation can be shown by induction on the structure of expression.

Theorem 3.1 *If* $\Gamma \vdash M : \tau$ *then* $\mathcal{T}_1\llbracket\Gamma\rrbracket \vdash \mathcal{W}_1\llbracket M\rrbracket : \mathcal{T}_1\llbracket\tau\rrbracket$.

3.4.2 Type flattening transformation

The preceding transformation changes the type of the deforested recursive function to $\tau \to (\langle \overrightarrow{h} : (\overrightarrow{k} : \delta)\rangle \Rightarrow \langle \overrightarrow{h} : (\overrightarrow{j} : \delta)\rangle) \to (\langle \overrightarrow{l} : (\overrightarrow{k} : \delta)\rangle \Rightarrow \langle \overrightarrow{l} : (\overrightarrow{j} : \delta)\rangle)$. By the definition of the preceding transformation, we can assume, for every label $i \in \overrightarrow{h} \cup \overrightarrow{l}$, that every expression $\mathrm{out}i\,M$ has type $(\overrightarrow{j} : \delta)$ and that for every expression $\langle i = M\rangle$, M has type either $(\overrightarrow{h} : \delta)$ or $(\overrightarrow{j} : \delta)$.

$$\begin{aligned} \mathcal{W}_2\llbracket \langle d_p = M^{(\overrightarrow{i} : \delta)}\rangle \rrbracket = \\ (d_1\overrightarrow{i} = \Omega^{\delta}, \ldots, d_p\overrightarrow{i} = \#\overrightarrow{i}\ \mathcal{W}_2\llbracket M\rrbracket, \\ \ldots, d_n\overrightarrow{i} = \Omega^{\delta}) \\ \text{where } \overrightarrow{d} = \overrightarrow{h} \text{ or } \overrightarrow{d} = \overrightarrow{l}, d_p \in \overrightarrow{d}, \\ \text{and } \overrightarrow{i} = \overrightarrow{k} \text{ or } \overrightarrow{i} = \overrightarrow{j}\,. \end{aligned}$$

$$\mathcal{W}_2\llbracket \mathrm{out}i\,M \rrbracket = (\overrightarrow{j} = \#i\overrightarrow{j}\ \mathcal{W}_2\llbracket M\rrbracket)$$
$$\text{where } i \in \overrightarrow{h} \text{ or } i \in \overrightarrow{l}\,.$$

$$\begin{aligned} \mathcal{W}_2\llbracket \mathbf{fun}\,\langle i_1 = y : (\overrightarrow{k} : \delta)\rangle \to M_1; \\ \cdots\quad ; \\ \langle i_n = y : (\overrightarrow{k} : \delta)\rangle \to M_n \rrbracket = \\ \lambda z : (\overrightarrow{i}\overrightarrow{k} : \mathcal{T}_2\llbracket\delta\rrbracket). \\ (i_1\overrightarrow{j} = \#i_1\overrightarrow{j}\ ((\lambda y.\mathcal{W}\llbracket M_1\rrbracket)(\overrightarrow{k} = \#i_1\overrightarrow{k}\ z)), \\ \cdots, \\ i_n\overrightarrow{j} = \#i_n\overrightarrow{j}\ ((\lambda y.\mathcal{W}\llbracket M_n\rrbracket)(\overrightarrow{k} = \#i_n\overrightarrow{k}\ z))) \\ \text{where } \overrightarrow{i} = \overrightarrow{h} \text{ or } \overrightarrow{i} = \overrightarrow{l}\,. \end{aligned}$$

$$\begin{aligned} \mathcal{T}_2\llbracket \langle d_p : (\overrightarrow{i} : \delta)\rangle \rrbracket = (\overrightarrow{d}\overrightarrow{i} : \mathcal{T}_2\llbracket\delta\rrbracket) \\ \text{where } \overrightarrow{d} = \overrightarrow{h} \text{ or } \overrightarrow{d} = \overrightarrow{l}, d_p \in \overrightarrow{d}, \\ \text{and } \overrightarrow{i} = \overrightarrow{k} \text{ or } \overrightarrow{i} = \overrightarrow{j}\,. \end{aligned}$$

$$\begin{aligned} \mathcal{T}_2\llbracket \langle \overrightarrow{i} : (\overrightarrow{k} : \delta)\rangle \Rightarrow \langle \overrightarrow{i} : (\overrightarrow{j} : \delta)\rangle \rrbracket = \\ (\overrightarrow{i}\overrightarrow{k} : \delta) \to (\overrightarrow{i}\overrightarrow{j} : \delta) \\ \text{where } \overrightarrow{i} = \overrightarrow{h} \text{ or } \overrightarrow{i} = \overrightarrow{l}\,. \end{aligned}$$

Figure 5: Type flattening transformation

For any pair of labels l and h, we write lh for a new label that is obtained by concatenation. For notational convention, we would also write $l\overrightarrow{j}$ for a sequence of concatenated labels $lj_1, \ldots, lj_m$, and $\overrightarrow{l}\overrightarrow{j}$ for an elementwise product sequence of concatenated labels, *i. e.*, $l_1 j_1, \ldots, l_i j_k, \ldots, l_n j_m$.

Figure 5 gives the definition of the type flattening transformation $\mathcal{W}_2$, together with corresponding type transformation $\mathcal{T}_2$. In the definition, Ω^{τ} denotes a diverging expression of type τ, namely $\mathbf{fix}\,(\lambda x : \tau.x)$.

We have the following theorem.

Theorem 3.2 *If* $\Gamma \vdash M : \tau$ *then* $\mathcal{T}_2\llbracket\Gamma\rrbracket \vdash \mathcal{W}_2\llbracket M\rrbracket : \mathcal{T}_2\llbracket\tau\rrbracket$.

3.4.3 Higher-order removal transformation

The last step of transformation process converts the recursive function of type $((\overrightarrow{h}\overrightarrow{k} : \delta) \to (\overrightarrow{h}\overrightarrow{j} : \delta)) \to ((\overrightarrow{l}\overrightarrow{k} : \delta) \to (\overrightarrow{l}\overrightarrow{j} : \delta))$ into $(\overrightarrow{h}\overrightarrow{j} : \delta, \overrightarrow{l}\overrightarrow{k} : \delta) \to$

$(\overrightarrow{h}\,\overrightarrow{k} : \delta, \overrightarrow{l}\,\overrightarrow{j} : \delta)$.

This transformation is more difficult to formalize than the preceding ones because it is a non-local transformation in the sense that the result of transforming a subexpression may be used in a context different where the subexpression originally were.

Due to this mobility of program code fragments, we need a few conditions described below in order to ensure the correctness of transformation.

Linear use of functions For every function g of type $(\overrightarrow{h}\,\overrightarrow{k} : \delta) \to (\overrightarrow{h}\,\overrightarrow{j} : \delta)$ and every label $i \in \overrightarrow{h}\,\overrightarrow{k}$, there is one and only one application site $g\,(i = M, \ldots)$ such that M is different from the diverging expression Ω. Similar condition is required for every function g of type $(\overrightarrow{l}\,\overrightarrow{k} : \delta) \to (\overrightarrow{l}\,\overrightarrow{j} : \delta)$.

Portability Every expression that has been moved to a different context from the original one behaves in the same way as it would do in the original context.

The side conditions for the syntactic form $(*)$ mentioned in Section 2.3.3 is intended to force these requirements. Exclusion of functional abstraction constructs is intended to match the number of execution of function applications with that of syntactic occurrences of application sites, and the fixed set of free variables makes transferred expressions portable, in the sense that free variables in transferred expressions capture the same denotations as they did at the original position.

Assuming the above conditions, for every function application site $g\,(i = M, \ldots)$ in expressions $M, M_1, \ldots, M_n$, if g has type either of type $(\overrightarrow{h}\,\overrightarrow{k} : \delta) \to (\overrightarrow{h}\,\overrightarrow{j} : \delta)$ or $(\overrightarrow{l}\,\overrightarrow{k} : \delta) \to (\overrightarrow{l}\,\overrightarrow{j} : \delta)$, then g must be a variable v of either x, y, $\mathtt{s}_1, \ldots, \mathtt{s}_n$. We write $M_{v,i}$ for such expression M. $M_{v,i}$ is uniquely determined due to the linearity condition.

Figure 6 defines the higher-order removal transformation, where $\mathcal{W}_3$ defines transformations for the syntactic forms $(*)$ and $(*')$ in Section 3.4.3 with subsidiary transformation $\mathcal{W}''_3$. We assume $z_0, \ldots, z_n$ are fresh variables.

Let us define type transformation $\mathcal{T}'_3$ corresponding to the subsidiary transformation $\mathcal{W}''_3$ as follows.

$$
\begin{aligned}
\mathcal{T}'_3[\![(\overrightarrow{h}\,\overrightarrow{k} : \delta)]\!] &= (\overrightarrow{h}\,\overrightarrow{k} : \mathcal{T}'_3[\![\delta]\!], \overrightarrow{l}\,\overrightarrow{j} : \mathcal{T}'_3[\![\delta]\!])\\
\mathcal{T}'_3[\![(\overrightarrow{l}\,\overrightarrow{k} : \delta)]\!] &= (\overrightarrow{l}\,\overrightarrow{k} : \mathcal{T}'_3[\![\delta]\!], \overrightarrow{h}\,\overrightarrow{j} : \mathcal{T}'_3[\![\delta]\!])\\
\mathcal{T}'_3[\![(\overrightarrow{l}\,\overrightarrow{j} : \delta)]\!] &= (\overrightarrow{h}\,\overrightarrow{k} : \mathcal{T}'_3[\![\delta]\!], \overrightarrow{l}\,\overrightarrow{j} : \mathcal{T}'_3[\![\delta]\!])\\
\mathcal{T}'_3[\![(\overrightarrow{h}\,\overrightarrow{j} : \delta)]\!] &= (\overrightarrow{l}\,\overrightarrow{k} : \mathcal{T}'_3[\![\delta]\!], \overrightarrow{h}\,\overrightarrow{j} : \mathcal{T}'_3[\![\delta]\!])
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{W}_3[\![\lambda x_0 : (\overrightarrow{h}\,\overrightarrow{k} : \delta) \to (\overrightarrow{h}\,\overrightarrow{j} : \delta).\\
&\qquad \mathbf{let}\, x_1 = f\ N_1\ (\lambda y_1 : (\overrightarrow{h}\,\overrightarrow{k} : \delta).M_1);\\
&\qquad\qquad \cdots\\
&\qquad\quad x_n = f\ N_n\ (\lambda y_n : (\overrightarrow{h}\,\overrightarrow{k} : \delta).M_n)\\
&\qquad \mathbf{in}\ \lambda y_0 : (\overrightarrow{l}\,\overrightarrow{k} : \delta).M_0]\!] =\\
&\lambda z_0 : (\overrightarrow{h}\,\overrightarrow{j} : \mathcal{T}'_3[\![\delta]\!], \overrightarrow{l}\,\overrightarrow{k} : \mathcal{T}'_3[\![\delta]\!]).\\
&\quad \mathbf{let}\, z_1 = f\ N_1\ (\overrightarrow{h}\,\overrightarrow{j} = \#\overrightarrow{h}\,\overrightarrow{j}\ \mathcal{W}''_3[\![M_1]\!],\\
&\qquad\qquad \overrightarrow{l}\,\overrightarrow{k} = \mathcal{W}''_3[\![M_{x_1, \overrightarrow{l}\,\overrightarrow{k}}]\!]);\\
&\qquad\quad \cdots\\
&\qquad z_n = f\ N_n\ (\overrightarrow{h}\,\overrightarrow{j} = \#\overrightarrow{h}\,\overrightarrow{j}\ \mathcal{W}''_3[\![M_n]\!],\\
&\qquad\qquad \overrightarrow{l}\,\overrightarrow{k} = \mathcal{W}''_3[\![M_{x_n, \overrightarrow{l}\,\overrightarrow{k}}]\!])\\
&\quad \mathbf{in}\ (\overrightarrow{l}\,\overrightarrow{j} = \#\overrightarrow{l}\,\overrightarrow{j}\ \mathcal{W}''_3[\![M_0]\!],\\
&\qquad \overrightarrow{h}\,\overrightarrow{k} = \mathcal{W}''_3[\![M_{x_0, \overrightarrow{h}\,\overrightarrow{k}}]\!])
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{W}_3[\![\mathbf{let}\, x_1 = f\ N_1\ (\lambda y_1 : (\overrightarrow{h}\,\overrightarrow{k} : \delta).M_1);\\
&\qquad\quad \cdots\\
&\qquad x_n = f\ N_n\ (\lambda y_n : (\overrightarrow{h}\,\overrightarrow{k} : \delta).M_n)\\
&\quad \mathbf{in}\ M_0]\!] =\\
&\quad \mathbf{let}\, z_1 = f\ N_1\ (\overrightarrow{h}\,\overrightarrow{j} = \#\overrightarrow{h}\,\overrightarrow{j}\ \mathcal{W}''_3[\![M_1]\!],\\
&\qquad\qquad \overrightarrow{l}\,\overrightarrow{k} = \mathcal{W}''_3[\![M_{x_1, \overrightarrow{l}\,\overrightarrow{k}}]\!]);\\
&\qquad\quad \cdots\\
&\qquad z_n = f\ N_n\ (\overrightarrow{h}\,\overrightarrow{j} = \#\overrightarrow{h}\,\overrightarrow{j}\ \mathcal{W}''_3[\![M_n]\!],\\
&\qquad\qquad \overrightarrow{l}\,\overrightarrow{k} = \mathcal{W}''_3[\![M_{x_n, \overrightarrow{l}\,\overrightarrow{k}}]\!])\\
&\quad \mathbf{in}\ \mathcal{W}''_3[\![M_0]\!]
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{W}''_3[\![x_0]\!] = \lambda z : (\overrightarrow{h}\,\overrightarrow{k} : \mathcal{T}'_3[\![\delta]\!], \overrightarrow{l}\,\overrightarrow{j} : \mathcal{T}'_3[\![\delta]\!]).z_0\\
&\mathcal{W}''_3[\![x_i]\!] = \lambda z : (\overrightarrow{h}\,\overrightarrow{k} : \mathcal{T}'_3[\![\delta]\!], \overrightarrow{l}\,\overrightarrow{j} : \mathcal{T}'_3[\![\delta]\!]).z_i\ (i \neq 0)\\
&\mathcal{W}''_3[\![y_i]\!] = z_i\\
&\mathcal{W}''_3[\![(\overrightarrow{p} = \overrightarrow{M})]\!] = (\overrightarrow{p} = \overrightarrow{\mathcal{W}''_3[\![M]\!]}, \overrightarrow{q} = \Omega^{\mathcal{T}'_3[\![\delta]\!]})\\
&\qquad \text{where } (\overrightarrow{p}, \overrightarrow{q}) \text{ is either } (\overrightarrow{h}\,\overrightarrow{k}, \overrightarrow{l}\,\overrightarrow{j}),\\
&\qquad (\overrightarrow{l}\,\overrightarrow{j}, \overrightarrow{h}\,\overrightarrow{k}), \quad (\overrightarrow{l}\,\overrightarrow{k}, \overrightarrow{h}\,\overrightarrow{j}), \quad \text{or}\\
&\qquad (\overrightarrow{h}\,\overrightarrow{j}, \overrightarrow{l}\,\overrightarrow{k}).
\end{aligned}
$$

Figure 6: Higher-order removal

Then we can show the type correctness of $\mathcal{W}''_3$.

Lemma 3.1 *Let Γ be a type environment such that $\Gamma(x_0) = (\overrightarrow{h}\,\overrightarrow{k} : \delta) \to (\overrightarrow{h}\,\overrightarrow{j} : \delta)$, $\Gamma(y_0) = (\overrightarrow{l}\,\overrightarrow{k} : \delta)$, and for every $i (1 \leq i \leq n)$ $\Gamma(x_i) = (\overrightarrow{l}\,\overrightarrow{k} : \delta) \to (\overrightarrow{l}\,\overrightarrow{j} : \delta)$ and $\Gamma(y_i) = (\overrightarrow{h}\,\overrightarrow{k} : \delta)$. If $\Gamma \vdash M : \tau$ and M has no occurrence of variables other than x'_is and y_i's, then we* *have* $\mathcal{T}_3'[\![\Gamma]\!] \cup z_0 : (\overrightarrow{h}\overrightarrow{j} : \delta, \overrightarrow{l}\overrightarrow{k} : \delta), z_1 : (\overrightarrow{h}\overrightarrow{k} : \delta, \overrightarrow{l}\overrightarrow{j} : \delta), \ldots, z_n : (\overrightarrow{h}\overrightarrow{k} : \delta, \overrightarrow{l}\overrightarrow{j} : \delta), \vdash \mathcal{W}_3'[\![M]\!] : \mathcal{T}_3'[\![\tau]\!]$.

The next theorem follows from this lemma.

Theorem 3.3 *Let N_1 and N_2 be the argument expressions of $\mathcal{W}_3$ in the first and second defining equation of Figure 6, respectively. If $\Gamma \cup f : \sigma \to \tau \vdash N_1 : \tau$ where $\tau = ((\overrightarrow{h}\overrightarrow{k} : \delta) \to (\overrightarrow{h}\overrightarrow{j} : \delta)) \to ((\overrightarrow{l}\overrightarrow{k} : \delta) \to (\overrightarrow{l}\overrightarrow{j} : \delta))$, then $\Gamma \cup f : \sigma \to \tau' \vdash \mathcal{W}_3[\![N_1]\!] : \tau'$ holds where $\tau' = (\overrightarrow{h}\overrightarrow{j} : \delta, \overrightarrow{l}\overrightarrow{k} : \delta) \to (\overrightarrow{h}\overrightarrow{k} : \delta, \overrightarrow{l}\overrightarrow{j} : \delta)$. Also, if $\Gamma \cup f : \sigma \to \tau \vdash N_2 : \tau$, then $\Gamma \cup f : \sigma \to \mathcal{T}_3'[\![\tau]\!] \vdash \mathcal{W}_3[\![N_2]\!] : \mathcal{T}_3'[\![\tau]\!]$ holds.*

4 Comparison with Macro Tree Transducer Composition Method

A precise and detailed analysis on the ability of the attribute grammar composition method [7] has been given by Giegerich [8]. He showed that a *single-use* condition, which restricts every use of attribute value to only once use, is essential for the composition to be successful, whereas the condition can be slightly relaxed in some cases.

The composition method for macro tree transducers proposed by Voigtländer and Kühnemann [18, 20, 19] applies under a condition that significantly relaxes the single-use condition: It requires, when applied to a composition of a function f_1 followed by f_2, f_1 to be *context linear* and f_2 to be *recursion linear*. A function f is context linear if every argument supplied to f (except for the recursion argument, on which f is defined inductively) is used only once in the function body of f; A function f is recursion linear if its function body has only a single recursive call to every different immediate substructure of the recursion argument. Since attribute grammars cannot express non-recursion linear functions, the relaxed condition enlarges the class of composable programs substantially.

Our deforestation method also works under the relaxed condition. Suppose we have the following program that we would like to deforest.

```
data T = Leaf | Node (T, T)

ex :: T → (h:: T) → (ex:: T)
ex (Node (t1,t2)) (h=h) =
  ex t1 (h=Node(#ex (ex t1 (h=h)),
                #ex (ex t2 (h=Leaf))))
ex Leaf (h=h) = (ex=h)

sp :: T → (k:: T) → (sp:: T)
sp (Node(t1,t2)) (k=k) =
  (sp= Node(#sp (sp t1 (k=k)),
            #sp (sp t2 (k=k))))
sp Leaf (k=k) = (sp=Node(k,k))

spex :: T → T
spex t =
  #sp (sp (#ex (ex t (h=Leaf))) (k=Leaf))
```

The function `spex` is a composition of functions `ex` followed by `sp`, where the former is context linear (but not recursion linear since there are two recursive calls to `t1`, the left subtree of binary node) and the latter is recursion linear (but not context linear since the argument `k` is used twice). Note that this example is selected just for demonstrating the compositional power of our deforestation method and it is intended no practical use.

Applying our deforestation method, we obtain the following result.

```
se :: T → (hsp:: T, exk :: T)
       → (hk:: T, exsp :: T)
se (Node(t1,t2)) inh =
  let s1l = se t1
       (hsp=Node(#exsp s1r, #exsp s2),
        exk=#exk inh)
      s1r = se t1
       (hsp=#hsp inh, exk=#hk s1l)
      s2  = se t2
       (hsp=Node(#hk s1r, #hk s1r),
        exk=#hk s1l)
  in (exsp=#exsp s1l, hk=#hk s1r)
se Leaf inh =
  (exsp=#hsp inh, hk=#exk inh)

aspex :: T → T
spex t =
  let s = se t (hsp=Node(#hk s, #hk s),
                exk=Leaf)
  in #exsp s
```

This evidences that our deforestation method is as powerful as Voigtländer and Kühnemann's, and of course it is more powerful than the attribute grammar composition method. Our transformation succeeds because the linearity condition imposed for the higher-order removal transformation (Section 2.3.3) permits programs to be non-recursion linear. Recursive calls to the same substructure are permitted, as far as such recursive calls are syntactically distinguished. (The **`let`** construct in the syntactic forms $(*)$ and $(*')$ in Section 3.4.3 identifies them by the variables $\mathtt{x}_1, \ldots, \mathtt{x}_n$.)

5 Conclusion and Future Work

We have proposed a type-driven transformation procedure for deforesting functions with accumulating parameters. The proposed deforestation method can deal with more programs than the existing composition method that has been developed for attribute grammars. It also provides an exposition on the attribute grammar composition algorithm that is more accessible from the side of functional programming community.

The present paper does not give a formal proof on the correctness of each transformation process, leaving it to intuitions. We believe that the correctness of our deforestation method can be proved by using Pitts' syntactic formulation of logical relation [14]. Since our transformation is driven by type information, the proof would be more comprehensive than that for the original attribute grammar composition method [6].

Finally to note, it would be very interesting to investigate the relationship between the macro tree transducer composition method and ours more closely. The example given in the previous section indicates that their composition abilities are very close, but it is an open question if they exactly coincide.

Acknowledgment I would like to thank Janis Voigtländer for an interesting discussion on the similarity between his macro tree transducer-based method and the present work.

References

1. R. S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21:239–250, 1984.
2. R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of ACM*, 24(1):44–67, 1977.
3. W.-N. Chin. Safe fusion of functional expressions II: Further improvemnts. *Journal of Functional Programming*, 4(4):515–555, 1994.
4. L. Correnson, E. Duris, D. Parigot, and G. Roussel. Declarative program transformation: A deforestation case-study. In *Principles and Practice of Declarative Programming*, volume 1702 of *LNCS*, pages 360–377. Springer Verlag, 1999.
5. J. Engelfriet and H. Vogler. Macro tree transducers. *Journal of Computer and System Sciences*, 31:71–146, 1985.
6. H. Ganzinger. Increasing modularity and language-independency in automatically generated compilers. *Science of Computer Programming*, 3(3):223–278, 1983.
7. H. Ganzinger and R. Giegerich. Attribute coupled grammars. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, volume 19(6) of *SIGPLAN Notices*, pages 157–170, June 1984.
8. R. Giegerich. Composition and evaluation of attribute coupled grammars. *Acta Informatica*, 25(4):355–423, May 1988.
9. A. Gill, J. Launchbury, and S. L. P. Jones. A short cut to deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 223–232, New York, NY, USA, June 1993. ACM Press.
10. The Haskell home page. `http://www.haskell.org/`.
11. T. Johnsson. Attribute grammars as a functional programming paradigm. In G. Kahn, editor, *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, volume 274 of *LNCS*, pages 154–173. Springer Verlag, 1987.
12. K. Kakehi, R. Glück, and Y. Futamura. On deforesting parameters of accumulating maps. In *Logic Based Program Synthesis and Transformation, 11th International Workshop, LOPSTR 2001*, volume 2372 of *LNCS*, pages 46–56. Springer Verlag, 2001.
13. D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.
14. A. Pitts. Parametric polymorphism and operational semantics. *Mathematical Structures in Computer Science*, 10(3):321–359, 2000.
15. T. Sheard and L. Fegaras. A fold for all seasons. In *Proceedings 6th ACM SIGPLAN/SIGARCH Int. Conf. on Functional Programming Languages and Computer Architecture, FPCA'93, Copenhagen, Denmark, 9–11 June 1993*, pages 233–242. ACM Press, New York, 1993.
16. J. Svenningsson. Shortcut fusion for accumulating

parameters & zip-like functions. In *Proc. of the 2002 International conference on functional programming*, 2002.

17. A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *Proceedings of the 7^th International Conference on Functional Programming Languages and Computer Architecture*, pages 306–313, La Jolla, California, June 1995. ACM SIGPLAN/SIGARCH and IFIP WG2.8, ACM Press.
18. J. Voigtländer. Composition of restricted macro tree transducers. Master's thesis, Dresden University of Technology, Germany, Mar. 2001.
19. J. Voigtländer. Using circular programs to deforest in accumulating parameters. In K. Asai and W.-N. Chin, editors, *ASIAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 126–137. ACM Press, 2002.
20. J. Voigtländer and A. Kühnemann. Composition of functions with accumulating parameters. Technical Report TUD-FI01-08, Dresden University of Technology, Germany, 2001.
21. P. Wadler. Theorems for free! In *Fourth International Conference on Functional Programming and Computer Architecture*, pages 347–359. Addison-Wesley, 1989.
22. P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, June 1990.

Appendix

A Correctness of Shortcut Fusion

We prove the correctness of the rewrite rule for shortcut fusion in Section 2.2, along the line of [9]. Here we only prove the case of lists, which should generalize to arbitrary algebraic data structures, similarly as in [17].

Theorem A.1 *If g is a function of type*

$$\begin{aligned}&\forall\beta.((A \to \beta \to \beta) \to \beta\\ &\quad \to (l_1 :: \beta, \cdots, l_n :: \beta) \to (j_1 :: \beta, \cdots, j_m :: \beta))\end{aligned}$$

where $n, m \geq 0$ and A is a constant type, then

$$\begin{aligned}&\mathtt{foldr}\, k\, z\, (\#j_q\, ((\mathtt{build}\, g)(l_1 = e_1, \cdots, l_n = e_n)))\\ &= \#j_q\, ((g\, k\, z)\\ &\qquad (l_1 = \mathtt{foldr}\, k\, z\, e_1, \cdots, l_n = \mathtt{foldr}\, k\, z\, e_n))\end{aligned}$$

holds for any q ($1 \leq q \leq m$).

Proof. The condition of the free theorem [21] states that:

For all $f : A \to B \to B$, $f' \in A \to B' \to B'$, and $h : B \to B'$, we have

$$\begin{aligned}&\forall a : A, \forall b : B,\\ &h(f\ a\ b) = f'\ a\ (h\ b)\\ &\Longrightarrow\ \forall b, b_1, \cdots, b_n : B,\\ &\qquad h(\#j_q\ (g_B\ f\ b\ (l_1 = b_1, \cdots, l_n = b_n)))\\ &\qquad = \#j_q(g'_B\ f'\ (h\ b)(l_1 = h\ b_1, \cdots, l_n = h\ b_n)),\end{aligned}$$

where g_B and $g_{B'}$ are the instances of g at B and B', respectively.

Instantiating this equation with $h = \mathtt{foldr}\ k\ z$, $f = $ Cons, and $f' = k$, we obtain:

$$\begin{aligned}&\forall a : A, \forall b : B,\\ &(\mathtt{foldr}\ k\ z)(f\ a\ b) = k\ a\ (\mathtt{foldr}\ k\ z\ b)\\ &\Longrightarrow \forall b, b_1, \cdots, b_n : B,\\ &\quad \mathtt{foldr}\ k\ z\ (\#j_q\ (g_B\ \mathrm{Cons}\ b\ (l_1 = b_1, \cdots, l_n = b_n)))\\ &\quad = \#j_q(g'_B\ k\ (\mathtt{foldr}\ k\ z\ b)\\ &\qquad\qquad (l_1 = \mathtt{foldr}\ k\ z\ b_1,\\ &\qquad\qquad\quad \cdots, l_n = \mathtt{foldr}\ k\ z\ b_n)).\end{aligned}$$

The premise part of the implication holds from the definition of `foldr`. Hence we obtain the following equation with instantiations $b = $ Nil and $b_i = e_i$ for every i.

$$\begin{aligned}&\mathtt{foldr}\ k\ z\ (\#j_q\ (g_B\ (\mathrm{Cons})\ \mathrm{Nil}\ (l_1 = e_1, \cdots, l_n = e_n)))\\ &= \#j_q(g'_B\ k\ (\mathtt{foldr}\ k\ z\ \mathrm{Nil})\\ &\qquad (l_1 = \mathtt{foldr}\ k\ z\ e_n, \cdots, l_n = \mathtt{foldr}\ k\ z\ e_n)).\end{aligned}$$

The result follows from the definition of `foldr` and `build`. □