Theoretical Computer Science 73 (1990) 231-248
North-Holland

DEFORESTATION: TRANSFORMING PROGRAMS TO ELIMINATE TREES *

Philip WADLER

Department of Computer Science, University of Glasgow, Glasgow G12 8QQ, UK

Abstract. An algorithm that transforms programs to eliminate intermediate trees is presented. The algorithm applies to any term containing only functions with definitions in a given syntactic form, and is suitable for incorporation in an optimizing compiler.

0. Introduction

Intermediate lists, and, more generally, intermediate trees, are both the basis and the bane of a certain style of programming in functional languages. For example, to compute the sum of the squares of the numbers from 1 to n, one could write the following program:

$$sum(map\ square(upto\ 1\ n)) \tag{1}$$

A key feature of this style is the use of functions (*upto*, *map*, *sum*) to encapsulate common patterns of computation ("consider the numbers from 1 to n", "apply a function to each element", "sum a collection of elements").

Intermediate lists are the basis of this style—they are the glue that holds the functions together. In this case, the list $[1, 2, \ldots, n]$ connects *upto* to *map*, and the list $[1, 4, \ldots, n^2]$ connects *map* to *sum*.

But intermediate lists are also the bane—they exact a cost at run time. If strict evaluation is used, the program requires space proportional to n. If lazy evaluation is used, space is not a problem: each list element is generated as it is needed, and list consumers and producers behave as co-routines. But even under lazy evaluation, each list element requires time to be allocated, to be examined, and to be de-allocated. Transforming the above to eliminate the intermediate lists gives

```
h 0 1 n
where
h a m n = if m > n
              then a
              else h(a + square m)(m + 1)n.                    (2)
```

This program is more efficient, regardless of the evaluation order, because all operations on list cells have been eliminated.

* An earlier version of this paper appeared in *European Symposium on Prog ·mming*, Nancy, France, March 1988, Lecture Notes in Computer Science 300 (Springer, Berlin, 1988).

This paper presents an algorithm that transforms programs to eliminate intermediate lists (and intermediate trees) called the Deforestation Algorithm. A form of function definition that uses no intermediate trees is characterized, called *treeless form*. An algorithm is given that can transform any term composed of functions in treeless form into a function that is itself in treeless form. For example, *sum*, *map square*, and *upto* all have treeless definitions, and applying the algorithm to program (1) yields a program equivalent to (2).

The algorithm appears suitable for inclusion in an optimizing compiler. Treeless form is easy to identify syntactically, and the transformation applies to any term (or sub-term) composed of treeless functions. The algorithm is most suitable for use with a lazy language; it also works for a strict language, but in this case can have the embarrassing effect of transforming an undefined program into one that returns a value.

Treeless form and the Deforestation Algorithm are presented in three steps. The first step presents "pure" treeless form in a first-order lazy functional language; in this form, no intermediate values whatsoever are allowed. This is too restrictive for most practical uses, so the second step extends treeless form by allowing one to use "blazing" (marking of trees according to type) to indicate where intermediate values may remain. Finally, the third step extends the results to some higher-order functions, by treating such functions as macros. These "higher-order macros" may also be of use in other applications.

A prototype of the transformer has been implemented in [4]; it was added to Augustsson and Johnsson's LML compiler [2]. The prototype handles blazed treeless form, and demonstrates that the transformer does work in practice. However, a thorough evaluation of the utility of these ideas must await an implementation that handles higher-order functions (as macros or otherwise).

This paper is the outgrowth of previous work on "listlessness"—transformations that eliminate intermediate lists [12, 13]. The new approach includes several improvements. First, the definition of treeless form is simpler than the definition of listless form. Second, the Deforestation Algorithm applies to *all* terms composed solely of treeless functions, whereas the corresponding algorithm in [13] applies only when a semantic condition, pre-order traversal, can be verified. Third, the treeless transformer is source-to-source (it converts functional programs into functional programs), whereas the listless transformer is not (it converts functional programs into imperative "listless programs"). However, the class of treeless functions is not the same as the class of listless functions. In some ways it is more general (it allows functions on trees, such as the *flip* function defined later), but in other ways it is more restricted (it does not apply to terms that traverse a data structure twice, such as *sum xs*/*length xs*). Whereas listless functions must evaluate in constant bounded space, treeless functions may use space bounded by the depth of the tree.

Much work on program transformation is based on a simple loop: instantiate/unfold/simplify/fold. This idea is found in the seminal work of Burstall and Darlington [3], in Turchin's "supercompiler" [11], and in the previous work on listlessness. It is equally central to deforestation, except that the instantiate step has been replaced by manipulation of **case** terms; this simplifies the bookkeeping, and also simplifies the presentation of the algorithm. A variant of deforestation using instantiation in place of **case** terms is investigated in [6].

The remainder of this paper is organized as follows. Section 1 describes the first-order language. Section 2 introduces treeless form. Section 3 outlines the Deforestation Algorithm and sketches a proof of its correctness. Section 4 extends treeless form to include blazing. Section 5 describes how to treat some higher-order functions as macros. Section 6 concludes.

1. Language

A first-order language with the following grammar is used:

$t ::= v$	variable
$\mid c\, t_1 \ldots t_k$	constructor application
$\mid f\, t_1 \ldots t_k$	function application
$\mid \textbf{case}\ t_0\ \textbf{of}\ p_1 : t_1 \mid \cdots \mid p_n : t_n$	case term
$p ::= c\, v_1 \ldots v_k$	pattern.

In an application, $t_1, \ldots, t_k$ are called the *arguments*, and in a **case** term, t_0 is called the *selector*, and $p_1 : t_1, \ldots, p_n : t_n$ are called the *branches*. Function definitions have the form

$$f\, v_1 \ldots v_k = t.$$

Example definitions are shown in Fig. 1.

The patterns in **case** terms may not be nested. Methods to transform **case** terms with nested patterns to ones without nested patterns are well known [1, 14].

```
list α          ::=  Nil | Cons α (list α)
tree α          ::=  Leaf α | Branch (tree α) (tree α)

append          :    list α → list α → list α
append xs ys    =    case xs of
                       Nil        :  ys
                       Cons x xs  :  Cons x (append xs ys)

flip            :    tree α → tree α
flip zt         =    case zt of
                       Leaf z          :  Leaf z
                       Branch xt yt    :  Branch (flip yt) (flip xt)
```

Fig. 1. Example definitions.

The language is typed using the Hindley-Milner polymorphic type system [8, 9, 5], found in LML and other languages; the reader is assumed to be familiar with this type system.

Each constructor c and function f has a fixed arity k. For example, the constructor *Nil* has arity 0, the constructor *Cons* has arity 2, and the function *flip* has arity 1. Although the language is first-order, terms and types are written in the same notation as for a higher-order language, to facilitate the extension in Section 5.

Traditionally, a term is said to be *linear* if no variable appears in it more than once. For example, (*append xs* (*append ys zs*)) is linear, but (*append xs xs*) is not. This definition must be extended slightly for linear **case** terms: no variable may appear in both the selector and a branch, although a variable may appear in more than one branch. For example, the definition of *append* is linear, even though *ys* appears in each branch.

The intended operational semantics of the language is normal order (leftmost outermost first) graph reduction. One term is said to be as efficient as another if, for every possible instantiation of the free variables, the first requires no more steps to reduce than the second.

2. Treeless form

Let F be a set of function names. A term is *treeless* with respect to F if it is linear, it only contains functions in F, and every argument of a function application and every selector of a **case** term is a variable.

In other words, writing tt for treeless terms with respect to F,

$$\begin{aligned} tt ::= {} & v \\ & \mid c\ tt_1 \ldots tt_k \\ & \mid f\ v_1 \ldots v_k \\ & \mid \textbf{case}\ v_0\ \textbf{of}\ p_1 : tt_1 \mid \cdots \mid p_n : tt_n \end{aligned}$$

where, in addition, tt is linear and each f is in F.

A collection of function definitions F is *treeless* if each right-hand side in F is treeless with respect to F. The definitions of *append* and *flip* in Fig. 1 are both treeless.

What is the rationale for this definition? The restriction that every argument of a function or selector of a **case** term must be a variable guarantees that no intermediate trees are created. It outlaws terms such as

flip (*flip zt*)

where (*flip zt*) returns an intermediate tree. On the other hand, constructor applications are not subject to the same restrictions. This allows terms such as

Branch (*flip yt*) (*flip xt*)

where the trees returned by (*flip yt*) and (*flip xt*) are not intermediate: they are part of the result.

The linearity restriction guarantees that certain program transformations do not introduce repeated computations. Burstall and Darlington use the term *unfolding* to describe the operation of replacing an instance of the left-hand side of an equation by the corresponding instance of the right-hand side [3]. Unfolding a definition with a non-linear right-hand side risks duplicating a term that is expensive to compute, making the program less efficient. For instance, a classic example of a non-linear function is

$$\mathit{square}\ x = x \times x.$$

If t is some term that is expensive to compute, it is preferable for a program to contain *square* t rather than its unfolded equivalent $t \times t$. On the other hand, an alternative definition is

$$\mathit{square}\ x = \exp(2 \times \log x).$$

Now *square* is linear, and there is no harm in unfolding *square* t to get $\exp(2 \times \log t)$. Insisting that treeless definitions are linear guarantees that they can be unfolded without sacrificing efficiency.

Loss of sharing can sometimes be avoided by using **let** terms. For example, with the first definition of *square*, unfolding *square* t using **let** yields

$$\textbf{let}\ x = t\ \textbf{in}\ x \times x.$$

This term is not treeless, because it contains t as an intermediate "tree", but in this case the intermediate "tree" is an integer, and so is harmless. Section 4 extends treeless form to allow intermediate terms of some types (such as *int*), and uses **let** terms for this purpose. But for types where it is desired to eliminate intermediate terms (such as *list* α and *tree* α) using **let** does not help: the whole point of unfolding is to bring together the argument with the function body to allow further transformations, and using **let** defeats this purpose.

Being treeless is a property of a definition, not of the function defined. Figure 2 gives two definitions of the function *flatten*. The definition of $\mathit{flatten}_1$ is treeless, while the definition of $\mathit{flatten}_0$ is not. (Unfortunately, the function to flatten a tree, rather than a list of lists, has no treeless definition; but see Section 6.)

It is now possible to present the main result.

Theorem 2.1 (Deforestation Theorem). *Every composition of functions with treeless definitions can be effectively transformed to a single function with a treeless definition, without loss of efficiency.*

The algorithm that carries out the effective transformation will be called the Deforestation Algorithm. The input to the Deforestation Algorithm is a linear term consisting of variables and functions with treeless definitions. The output is an equivalent treeless term and a (possibly empty) collection of treeless definitions. Although the statement above only guarantees no loss of efficiency, there will in fact be a gain whenever the original term contains an intermediate tree.

```
flatten_0        : list (list α) → list α
flatten_0 xss    = case xss of
                     Nil          : Nil
                     Cons xs xss  : append xs (flatten_0 xss)

flatten_1        : list (list α) → list α
flatten_1 xss    = case xss of
                     Nil          : Nil
                     Cons xs xss  : flatten_1' xs xss

flatten_1'       : list α → list (list α) → list α
flatten_1' xs xss = case xs of
                     Nil          : flatten_1 xss
                     Cons x xs    : Cons x (flatten_1' xs xss)
```

Fig. 2. A non-treeless and a treeless definition of *flatten*.

Figure 3 gives two examples of applying the Deforestation Algorithm, to the compositions *append* (*append xs ys*) *zs* and *flip* (*flip zt*). If m is the length of *xs* and n is the length of *ys*, then the original append term takes time $2m+n$ to compute, whereas the transformed version takes time $m+n$ to compute. The transformation introduces two new (treeless) definitions, h_0 and h_1; observe that h_1 is equivalent to *append*. Incidentally, *append xs* (*append ys zs*) is transformed into exactly the same term, modulo renaming; so, as a by-product, the Deforestation Algorithm provides a proof that *append* is associative.

The characterization of treeless definitions is purely syntactic, so it is easy for the user to determine when deforestation applies. The user need not be familiar with the details of the Deforestation Algorithm itself.

3. The Deforestation Algorithm

The heart of the Deforestation Algorithm is the set of seven rules shown in Fig. 4. Write $T[\![t]\!]$ to denote the result of converting term t to treeless form. It is required that

$$t = T[\![t]\!].$$

That is, t and $T[\![t]\!]$ should compute the same value.

Simple examination shows that the rules cover all possible terms: of the four kinds of term (variable, constructor application, function application, **case** term) three are covered directly, and for **case** terms, all four possibilities for the selector are considered.

```
                    append (append xs ys) zs

                          transforms to

h0 xs ys zs
where
h0 xs ys zs  =  case xs of
                  Nil        :  h1 ys zs
                  Cons x xs  :  Cons x (h0 xs ys zs)
h1 ys zs     =  case ys of
                  Nil        :  zs
                  Cons y ys  :  Cons y (h1 ys zs)
```

```
                          flip (flip zt)

                          transforms to

h0 zt
where
h0 zt  =  case zt of
            Leaf z         :  Leaf z
            Branch xt yt   :  Branch (h0 xt) (h0 yt)
```

Fig. 3. Results of applying the Deforestation Algorithm.

It is clear that each of the rules preserves equivalence. In rules (1), (2), and (4), the basic form already matches treeless form, and the components are converted recursively. In rules (3) and (6), a function application is unfolded, yielding an equivalent term that is converted recursively. For rules (5) and (7), the **case** term is simplified, and the result is converted recursively. (Rule (7) is valid only if no variable in $p_1, \ldots, p_m$ occurs free in any of the branches $p'_1 : t'_1, \ldots, p'_n : t'_n$. It is always possible to rename the bound variables so that this condition applies.)

There is one problem: the algorithm as given does not always terminate! An example of applying rules (1)-(7) is shown in Fig. 5. This shows transformation of the term

flip (*flip zt*).

After the steps shown, the final term reached is

```
case zt of
  Leaf z        : Leaf z
  Branch xt yt  : Branch (T⟦flip (flip xt)⟧)(T⟦flip (flip yt)⟧).        (*)
```

$$
\begin{array}{lll}
(1) & T\llbracket v \rrbracket & = \; v \\[1ex]
(2) & T\llbracket c\ t_1\ \dots\ t_k \rrbracket & = \; c\ (T\llbracket t_1 \rrbracket)\ \dots\ (T\llbracket t_k \rrbracket) \\[1ex]
(3) & T\llbracket f\ t_1\ \dots\ t_k \rrbracket & = \; T\llbracket t[t_1/v_1, \dots, t_k/v_k] \rrbracket \\
 & & \quad \textbf{where } f \textbf{ is defined by } f\ v_1\ \dots\ v_k = t \\[1ex]
(4) & \multicolumn{2}{l}{T\llbracket \textbf{case}\ v\ \textbf{of}\ p'_1 : t'_1 \mid \cdots \mid p'_n : t'_n \rrbracket} \\
 & & = \; \textbf{case}\ v\ \textbf{of}\ p'_1 : T\llbracket t'_1 \rrbracket \mid \cdots \mid p'_n : T\llbracket t'_n \rrbracket \\[1ex]
(5) & \multicolumn{2}{l}{T\llbracket \textbf{case}\ c\ t_1\ \dots\ t_k\ \textbf{of}\ p'_1 : t'_1 \mid \cdots \mid p'_n : t'_n \rrbracket} \\
 & & = \; T\llbracket t'_i[t_1/v_1, \dots, t_k/v_k] \rrbracket \\
 & & \quad \textbf{where } p'_i = c\ v_1\ \dots\ v_k \\[1ex]
(6) & \multicolumn{2}{l}{T\llbracket \textbf{case}\ f\ t_1\ \dots\ t_k\ \textbf{of}\ p'_1 : t'_1 \mid \cdots \mid p'_n : t'_n \rrbracket} \\
 & & = \; T\llbracket \textbf{case}\ t[t_1/v_1, \dots, t_k/v_k]\ \textbf{of}\ p'_1 : t'_1 \mid \cdots \mid p'_n : t'_n \rrbracket \\
 & & \quad \textbf{where } f \textbf{ is defined by } f\ v_1\ \dots\ v_k = t \\[1ex]
(7) & \multicolumn{2}{l}{T\llbracket \textbf{case}\ (\textbf{case}\ t_0\ \textbf{of}\ p_1 : t_1 \mid \cdots \mid p_m : t_m)\ \textbf{of}\ p'_1 : t'_1 \mid \cdots \mid p'_n : t'_n \rrbracket} \\
 & & = \; T\llbracket \textbf{case}\ t_0\ \textbf{of} \\
 & & \qquad p_1 \; : \; (\textbf{case}\ t_1\ \textbf{of}\ p'_1 : t'_1 \mid \cdots \mid p'_n : t'_n) \\
 & & \qquad \dots \\
 & & \qquad p_m \; : \; (\textbf{case}\ t_m\ \textbf{of}\ p'_1 : t'_1 \mid \cdots \mid p'_n : t'_n) \rrbracket
\end{array}
$$

Fig. 4. Transformation rules for the Deforestation Algorithm.

This contains two renamings of the original expression, and so the same rules may be applied again without end.

The trick to avoiding this infinite regresss is to introduce appropriate *new* function definitions. The example above requires the introduction of a function h_0 that satisfies the equation

$$h_0\ zt = T\llbracket flip\ (flip\ zt) \rrbracket .$$

Now when the expansion of $T\llbracket flip\ (flip\ zt) \rrbracket$ reaches the form (*) above, the two occurrences of $T\llbracket \dots \rrbracket$ match the right-hand side of this equation. They can therefore be replaced by the corresponding left-hand side, giving

$$
\begin{array}{lll}
h_0\ zt = \textbf{case}\ zt\ \textbf{of} & & \\
\quad Leaf\ z & : & Leaf\ z \\
\quad Branch\ xt\ yt & : & Branch\ (h_0\ xt)\ (h_0\ yt).
\end{array}
$$

This completes the transformation; the term *flip* (*flip zt*) is equivalent to the treeless term $h_0\ zt$, where h_0 has the treeless definition just derived.

```
T⟦flip (flip zt)⟧

  = T⟦case (flip zt) of                                            (by (3))
        Leaf z        : Leaf z
        Branch xt yt  : Branch (flip yt) (flip xt)⟧

  = T⟦case (case zt of                                             (by (6))
               Leaf z'        : Leaf z'
               Branch xt' yt' : Branch (flip yt') (flip xt')) of
        Leaf z        : Leaf z
        Branch xt yt  : Branch (flip yt) (flip xt)⟧

  = T⟦case zt of                                                   (by (7))
        Leaf z        : (case Leaf z of
                           Leaf z'        : Leaf z'
                           Branch xt' yt' : Branch (flip yt') (flip xt'))
        Branch xt yt  : (case Branch (flip yt) (flip xt) of
                           Leaf z'        : Leaf z'
                           Branch xt' yt' : Branch (flip yt') (flip xt'))⟧

  = case zt of                                                     (by (4))
      Leaf z        : T⟦(case Leaf z of
                           Leaf z'        : Leaf z'
                           Branch xt' yt' : Branch (flip yt') (flip xt'))⟧
      Branch xt yt  : T⟦(case Branch (flip yt) (flip xt) of
                           Leaf z'        : Leaf z'
                           Branch xt' yt' : Branch (flip yt') (flip xt'))⟧

  = case zt of                                                (by (5), (5))
      Leaf z        : T⟦Leaf z⟧
      Branch xt yt  : T⟦Branch (flip (flip xt)) (flip (flip yt))⟧

  = case zt of                                           (by (2), (1), (2))
      Leaf z        : Leaf z
      Branch xt yt  : Branch (T⟦flip (flip xt)⟧) (T⟦flip (flip yt)⟧)
```

Fig. 5. Deforestation of *flip* (*flip zt*).

When should new definitions be introduced? Any infinite sequence of steps must contain applications of rules (3) or (6), the unfold rules. Therefore, it is sufficient to take as potential right-hand sides, each term of the form *T*⟦. . .⟧ encountered just before applying rules (3) or (6). Keep a list of all such terms. Whenever a term is encountered for a second time, create the appropriate function definition and replace each instance of the term by a corresponding call of the function. Note that it is sufficient if the new term is a *renaming* of a previous term. For example, in the transformation above, $(flip\,(flip\,xt))$ was a renaming of $(flip\,(flip\,zt))$, and was replaced by the corresponding call $(h_0\,xt)$. The new term must be a renaming, rather than a more general instance, of the previous term; this guarantees that the resulting function call has the form $(f\,v_1 \ldots v_k)$, and hence is a treeless term.

It is a simple inductive proof to show that if the computation of $T[\![t]\!]$ terminates then the resulting term is in treeless form, and this term will itself be equivalent to t. Define the *depth* of a term to be zero if it is a variable, and one greater than the maximum depth of its subterms otherwise. Below is sketched a proof that whenever t is a legal input to the Deforestation Algorithm, then there is a bound on the depth of the terms of the form $T[\![\ldots]\!]$ encountered while applying rules (1)-(7). Since the terms are bounded in depth, and there are only a finite number of constructor and function symbols involved, then there are only a finite number of different terms (modulo renaming). Thus, eventually a renaming of a previous term must be encountered, and the algorithm is guaranteed to terminate.

As mentioned previously, linearity guarantees that the unfold rules, (3) and (6), never introduce a repeated computation. It is easy to verify that the other rules also do not duplicate computations, and hence the derived treeless term is at least as efficient as the original term.

It remains to show that there is a bound on the depth of the terms encountered by the Deforestation Algorithm. Define the terms of *order o* as follows: a term of order 0 is a variable, and a term of order $o+1$ has the form

$$tt[tt_1^o/v_1, \ldots, tt_n^o/v_n]$$

where tt is a treeless term, $tt_1^o, \ldots, tt_n^o$ are terms of order o, and $v_1, \ldots, v_n$ are the free variables of tt. That is, a term of order o is an o-fold substitution of treeless terms; a term is treeless iff it is of order 1.

Lemma 3.1. *Applying each of rules* (1)-(7) *to a term of order o results in a term of order o.*

Proof. Induct on o, then use structural induction on the form of the term, one case for each of the rules (1)-(7). The cases for rules (3) and (5) are typical. Case (3): Since the arguments of a treeless function application are variables, a function application of order $o+1$ must have the form $f\,tt_1^o \ldots tt_k^o$, where the arguments are of order o. If f is defined by $f\,v_1 \ldots v_k = tt$, then applying rule (3) yields the term $tt[tt_1^o/v_1, \ldots, tt_k^o/v_k]$, which is of order $o+1$, as required. Case (5): Since the selector of a treeless **case** term is a variable, if the left-hand side of rule (5) is of order $o+1$ it must have the form

$$\textbf{case}\ c\,tt_1^o \ldots tt_k^o\ \textbf{of}\ p_1 : tt_1^{o+1} \mid \cdots \mid p_n : tt_n^{o+1}.$$

If p_i has the form $c\,v_1 \ldots v_k$, then tt_i^{o+1} will have the form $tt[t_1'^o/v_1', \ldots, tt_m'^o/v_m']$, where $v_1', \ldots, v_m'$ are the free variables of tt other than $v_1, \ldots, v_k$ (which cannot be substituted for, since they are bound). Applying rule (5) yields the term

$$tt[tt^o_1/v_1, \ldots, tt^o_k/v_k, tt'^o_1/v'_1, \ldots, tt'^o_m/v'_m]$$

which is of order $o+1$, as required. □

A legal input to the Deforestation Algorithm is a term consisting only of functions and variables. Such a term will have an order o equal to its depth. Let d be the maximum of 1 and the depths of all right-hand sides of function definitions referred to (directly or indirectly) from the input. It is straightforward to show that any term encountered by the Deforestation Algorithm has a depth bounded by $d \times o$. This guarantees that the Deforestation Algorithm terminates whenever it is applied to a legal input, and completes the proof of the Deforestation Theorem.

It may be useful to apply the Deforestation Algorithm to terms other than its legal inputs. In this case termination is not guaranteed, but when it does terminate the algorithm still returns an equivalent treeless term. For example, applying the algorithm to the non-treeless definition of $flatten_0$ in Fig. 2 yields (a renaming of) the treeless definition of $flatten_1$.

4. Blazed treeless form

The definition of treeless form given in the previous section, henceforth called *pure* treeless form, is quite restrictive. Consider the definition

```
upto     : int → int → list int
upto m n = case (m > n) of
               True  : Nil
               False : Cons m (upto (m + 1) n).
```

For example, *upto* 1 4 returns [1, 2, 3, 4]. (Infix notation is used for legibility; $m > n$ may be taken as equivalent to $(>)\ m\ n$, where $(>)$ is a function name, and similarly for $m+1$.)

This definition is not in pure treeless form: first, because it contains a selector $(m > n)$ and a function argument $(m+1)$ that are not variables; and, second, because it is not linear (m appears once in the selector and twice in the second branch). But in all cases, the offending intermediate "tree" is really an integer.

To accommodate definitions such as *upto*, all terms will be divided into two kinds, marked with either a $\oplus$ or a $\ominus$. In forestry, *blazing* is the operation of marking a tree by making a cut in its bark, so the mark $\oplus$ or $\ominus$ will be called the blazing of the term. The idea is that deforestation should eliminate ("fell") all intermediate terms ("trees") blazed $\oplus$, but that intermediate terms blazed $\ominus$ may remain. Blazing will be assigned solely on the basis of type, and all terms of the same type must be blazed the same way. In the following, all terms of type *list* or *tree* will be blazed $\oplus$, and all terms of type *int* or *bool* will be blazed $\ominus$. If the type of a term is a type variable, such as α, then it should also be blazed $\ominus$; this is sensible because a value is given a variable type only if its internal structure is never manipulated. Writing $t^{\oplus}$ indicates that t is of a type blazed $\oplus$, and writing $t^{\ominus}$ indicates that t is of a type blazed $\ominus$.

In the definition of pure treeless form, the places where intermediate values could potentially appear (function arguments and **case** selectors) are restricted to be variables, and terms are required to be linear. For *blazed treeless form*, the places where intermediate values could potentially appear are restricted either to be variables or to be blazed $\ominus$, and terms are required to be linear only in variables blazed $\oplus$.

This yields the following new grammar for treeless terms with respect to a set of function names F:

$$
\begin{aligned}
tt ::= {} & vv \\
& \mid (c\ tt_1 \ldots tt_k)^{\oplus} \\
& \mid (f\ vv_1 \ldots vv_k)^{\oplus} \\
& \mid (\textbf{case}\ vv_0\ \textbf{of}\ p_1 : tt_1 \mid \cdots \mid p_n : tt_n)^{\oplus} \\
vv ::= {} & v \\
& \mid (c\ vv_1 \ldots vv_k)^{\ominus} \\
& \mid (f\ vv_1 \ldots vv_k)^{\ominus} \\
& \mid (\textbf{case}\ vv_0\ \textbf{of}\ p_1 : vv_1 \mid \cdots \mid p_n : vv_n)^{\ominus}
\end{aligned}
$$

where in addition tt and vv are linear in variables blazed $\oplus$, and each f is in F. Note that $tt^{\ominus}$ is equivalent to $vv^{\ominus}$, and $vv^{\oplus}$ is equivalent to $v^{\oplus}$. As before, a collection of definitions F is treeless if each right-hand side in F is treeless with respect to F. The definition of *upto* and all the definitions in Fig. 6 are treeless.

The Deforestation Theorem carries over virtually unchanged.

Theorem 4.1 (Blazed Deforestation Theorem). *Every composition of functions with blazed treeless definitions can be effectively transformed to a single function with a blazed treeless definition, without loss of efficiency.*

Two examples of applying the Blazed Deforestation Algorithm are shown in Fig. 7.

To accommodate blazing, the Deforestation Algorithm is extended as follows. If during the course of transformation a sub-term arises that is blazed $\ominus$, this sub-term may be extracted and transformed independently. It is convenient to introduce the notation **let** $v^{\ominus} = t_0^{\ominus}$ **in** t_1 to represent the result of such an extraction. A **let** term will only be introduced through extraction, so the bound variable will always be blazed $\ominus$.

```
sum           :  list int → int
sum xs        =  sum' 0 xs

sum'          :  int → list int → int
sum' a xs     =  case xs of
                    Nil          :  a
                    Cons x xs    :  sum' (a + x) xs

squares       :  list int → list int
squares xs    =  case xs of
                    Nil          :  Nil
                    Cons x xs    :  Cons (square x) (squares xs)

sumtr         :  tree int → int
sumtr xt      =  case xt of
                    Leaf x       :  x
                    Branch xt yt :  sumtr xt + sumtr yt

squaretr      :  tree int → tree int
squaretr xt   =  case xt of
                    Leaf x       :  Leaf (square x)
                    Branch xt yt :  Branch (squaretr xt) (squaretr yt)
```

Fig. 6. More example definitions.

For example, applying extraction to the term

$$sum'\ 0\ (squares\ (upto\ 1\ n))$$

yields the term

$$\begin{array}{l} \textbf{let } u_0 = 0 \textbf{ in} \\ \quad \textbf{let } u_1 = 1 \textbf{ in} \\ \quad\quad sum'\ u_0\ (squares\ (upto\ u_1\ n)). \end{array}$$

Later in the same transformation, applying extraction to the term

$$sum'\ (u_0 + square\, x)\ (squares\ (upto\ (u_1 + 1)\ n))$$

yields the term

$$\begin{array}{l} \textbf{let } u_2 = u_0 + square\, x \textbf{ in} \\ \quad \textbf{let } u_3 = u_1 + 1 \textbf{ in} \\ \quad\quad sum'\ u_2\ (squares\ (upto\ u_3\ n)). \end{array}$$

The inner term here is a renaming of the inner term of the previous expression, and

```
                sum (squares (upto 1 n))

                     transforms to

h0 0 1 n
where
h0 u0 u1 n  =  case (u1 > n) of
                 True  : u0
                 False : h0 (u0 + square u1) (u1 + 1) n
```

```
                sumtr (squaretr xt)

                    transforms to

h0 xt
where
h0 xt  =  case xt of
            Leaf x        : square x
            Branch xt yt  : h0 xt + h0 yt
```

Fig. 7. Results of applying the Blazed Deforestation Algorithm.

will cause the appropriate new function to be defined:

$$h_0\ u_0\ u_1\ n = T[\![sum'\ u_0\ (squares\ (upto\ u_1\ n))]\!].$$

Calls to h_0 will now replace the inner terms above.

Extraction forces all arguments of a function blazed $\ominus$ to be variables. This is why it is not necessary for terms to be linear in variables blazed $\ominus$: since unfolding only replaces such variables by other variables, no duplication of a term that is expensive to compute can occur.

To the definition of $T[\![t]\!]$ in Fig. 4 must be added the four rules in Fig. 8. Rules (8) and (9) supersede rules (3) and (6), respectively, in the case where the result and all arguments of a function are blazed $\ominus$. In this case it is not necessary to unfold the application: it can be simply left in place unchanged. In particular, rules (8) and (9) cover all applications of primitive functions, such as $t_0 > t_1$ or $t_0 + t_1$, which cannot be unfolded anyway. Rules (10) and (11) manage occurrences of **let**. (Rule (11) is only valid if v does not occur free in any of the branches $p'_1 : t'_1, \ldots, p'_n : t'_n$. It is always possible to rename the bound variables so that this condition applies.)

After the transformation is complete, all terms of the form **let** $v^{\ominus} = tt_0^{\ominus}$ **in** tt_1 may be removed as follows. If v occurs at most once in tt_1, the term may be replaced

(8) $T[\![(f\ t_1^\ominus \ldots t_k^\ominus)^\ominus]\!]$
$= (f\ (T[\![t_1^\ominus]\!]) \ldots (T[\![t_k^\ominus]\!]))^\ominus$

(9) $T[\![\textbf{case}\ (f\ t_1^\ominus \ldots t_k^\ominus)^\ominus\ \textbf{of}\ p_1' : t_1' \mid \cdots \mid p_n' : t_n']\!]$
$= \textbf{case}\ (f\ (T[\![t_1^\ominus]\!]) \ldots (T[\![t_k^\ominus]\!]))^\ominus\ \textbf{of}\ p_1' : T[\![t_1']\!] \mid \cdots \mid p_n' : T[\![t_n']\!]$

(10) $T[\![\textbf{let}\ v^\ominus = t_0^\ominus\ \textbf{in}\ t_1]\!]$
$= \textbf{let}\ v^\ominus = T[\![t_0^\ominus]\!]\ \textbf{in}\ T[\![t_1]\!]$

(11) $T[\![\textbf{case}\ (\textbf{let}\ v^\ominus = t_0^\ominus\ \textbf{in}\ t_1)\ \textbf{of}\ p_1' : t_1' \mid \cdots \mid p_n' : t_n']\!]$
$= \textbf{let}\ v^\ominus = T[\![t_0^\ominus]\!]\ \textbf{in}\ T[\![\textbf{case}\ t_1\ \textbf{of}\ p_1' : t_1' \mid \cdots \mid p_n' : t_n']\!]$

Fig. 8. Additional rules for the Blazed Deforestation Algorithm.

by $tt_1[tt_0/v]$. If v occurs more than once, introduce a new function h defined by $h\,v = tt_1$, and replace the term by $h\,tt_0$. Since tt_0 is blazed $\ominus$, this application is a treeless term. (Alternatively, simply add **let** terms to the language, and extend the definition of treeless term to include terms in the above form.)

It is straightforward to extend the previous results to show that the modified Deforestation Algorithm satisfies the requirements of the Blazed Deforestation Theorem.

5. Higher-order macros

From the user's point of view, one of the most attractive features of programming in a functional style is the use of higher-order functions. However, for the implementor of a program transformation system, such as the Deforestation Algorithm, first-order languages may be easier to cope with. This section shows how much (but not all) of the expressiveness of higher-order functions can be achieved in a first-order language, by treating higher-order functions as macros. The same idea may be useful for a variety of applications where it is easier to deal with a first-order language but the power of a higher-order language is desirable. (Goguen has championed the notion that first-order languages often suffice, and that it is preferable to use them where possible [7]. He achieves an effect similar to higher-order macros by using parameterized modules in the first-order language OBJ.)

The first step is to add **where** terms to the language. These have the form

t **where** $d_1 ; \ldots ; d_n$

where t is a term and $d_1, \ldots, d_n$ are function definitions. This can be translated back into the equation language in a straightforward manner, by use of a technique called *lambda lifting* [2, 10]. In particular, if $d_1, \ldots, d_n$ contain no free variables then the term above is just equivalent to t, where the definitions $d_1, \ldots, d_n$ are added to the top-level list of definitions, with systematic renaming of functions (according to the scope of the **where** clause) to avoid any name conflicts.

The second step is to add higher-order macro definitions. These have the form

$$f\, v_1 \ldots v_k \triangleq t.$$

That is, they look like ordinary definitions, except with $\triangleq$ in place of $=$. The term t may now contain variables in place of function names, and applications are no longer restricted by arity. The Hindley-Milner type system is still used. The formal parameters $v_1, \ldots, v_n$ may now have a ground type, like *int* or (*list* α), or a function type, like (*int* $\rightarrow$ *int*), or even a higher-order type, like ((*int* $\rightarrow$ *int*) $\rightarrow$ *int*). The only restriction is that ***higher-order macros cannot be recursive.***

The lack of recursion, combined with the Hindley-Milner type discipline, guarantees that all higher-order definitions can be expanded out at compile time, with no risk of a non-terminating expansion. But at first the lack of recursion may seem overly restrictive. Does it not rule out higher-order functions such as *map* and *fold*? No, it does not, because first-order recursion is still possible using the **where** facility defined above. Definitions of *map* and *fold* are given in Fig. 9; recursion is limited to the first-order functions g and h.

With the definitions in Fig. 9, one can write terms such as

sum (*map square* (*upto* 1 *n*)),
map sum (*map* (*map square*) *xss*),
(*map square* ∘ *map cube*) *xs*,
map (*square* ∘ *cube*) *xs*.

```
map          :  (α → β) → list α → list β
map f xs     ≙  g xs
                where
                g xs  =  case xs of
                           Nil        :  Nil
                           Cons x xs  :  Cons (f x) (g xs)

fold         :  (α → β → α) → α → list β → α
fold f a xs  ≙  h a xs
                where
                h a xs  =  case xs of
                             Nil        :  a
                             Cons x xs  :  h (f a x) xs

sum          :  list int → int
sum          ≙  fold (+) 0

(∘)          :  (β → γ) → (α → β) → α → γ
(f ∘ g) x    ≙  f (g x)
```

Fig. 9. Example higher-order definitions.

Each of these expands out to a first-order program, which can then be transformed using the Deforestation Algorithm of the preceding sections.

The mechanism defined here covers many, but not all, uses of higher-order functions. In particular, all data structures are first-order, so it is not possible, for instance, to have a list of functions.

Higher-order macros provide one way to extend the Deforestation Algorithm from a first-order language, and they may be valuable for other applications as well. However, their worth is not yet proven. An alternative would be to formulate a version of the Deforestation Theorem that applies to higher-order functions directly, without the need to treat them as macros.

6. Conclusion

An oft-repeated justification for the study of functional programming is that functional programs are eminently suited for program transformation. And indeed, program transformation is a star member of the repertoire for writers of functional compilers. For example, many steps in the LML compiler involve transformation techniques [2]. Deforestation appears to be an attractive candidate for the next application of program transformation to compiler technology.

An important feature of the Deforestation Algorithm is that it is centred on an easily recognized class of definitions, treeless form. This eases the task of the compiler writer. Perhaps even more importantly, it eases the task of the compiler user, because it is easy to characterize what sort of expressions will be optimized and what sort of optimizations will be performed. Further work is desirable in two directions.

First, treeless form may be generalized. One possible generalization rests on the observation that some function arguments, such as the second argument to *append*, appear directly in the function result. These arguments might be treated in the same way as arguments to constructors in the definition of treeless form. It was previously noted that the function to flatten a tree has no treeless definition; with this generalization, it would. Related ideas are discussed in [15].

Second, further practical experience should be acquired, in order to assess better the utility of the ideas presented here.

Acknowledgment

I am grateful to Kei Davis for acting as a sounding board and undertaking to implement some of the ideas reported here, and to Cordelia Hall and Catherine Lyons for their comments on this paper. This work was in part performed at Oxford University under a research fellowship from ICL.

References

[1] L. Augustsson, Compiling pattern matching, in: *Proc. Conf. on Functional Programming Languages and Computer Architecture*, Nancy, France, Lecture Notes in Computer Science **201** (Springer, Berlin, 1985).

[2] L. Augustsson and T. Johnsson, The Chalmers Lazy ML compiler, *Comput. J.* **32**(2) (1989) 127-141.

[3] R.M. Burstall and J. Darlington, A transformation system for developing recursive programs *J. ACM* **24**(1) (1977) 44-67.

[4] M.K. Davis, Deforestation: transformation of functional programs to eliminate intermediate trees, M.Sc. Dissertation, Programming Research Group, Oxford University, 1987.

[5] L. Damas and R. Milner, Principal type schemes for functional programs, in: *Proc. ACM Symp. on Principles of Programming Languages* (1982).

[6] A.B. Ferguson and P.L. Wadler, When will deforestation stop? in: *Proc. Glasgow Workshop on Functional Programming*, Rothesay, Isle of Bute, August 1988, Research Report 89/R4, Department of Computing Science, University of Glasgow, 1989.

[7] J.A. Goguen, Higher order functions considered unnecessary for higher order programming, Technical Report SRI-CSL-88-1, SRI International, 1988.

[8] R. Hindley, The principal type scheme of an object in combinatory logic, *Trans. Amer. Math. Soc.* **146** (1979) 29-60.

[9] R. Milner, A theory of type polymorphism in programming, *J. Comput. System Sci.* **17** (1978) 348-375.

[10] S.L. Peyton Jones, *The Implementation of Functional Programming Languages* (Prentice Hall, Englewood Cliffs, NJ, 1987).

[11] V.F. Turchin, R. M. Nirenberg and D.V. Turchin, Experiments with a supercompiler, in: *Proc. ACM Symp. on Lisp and Functional Programming*, Pittsburgh, PA (1982).

[12] P.L. Wadler, Listlessness is better than laziness: lazy evaluation and garbage collection at compile time, in: *Proc. ACM Symp. on Lisp and Functional Programming*, Austin, TX (1984).

[13] P.L. Wadler, Listlessness is better than laziness II: composing listless functions, in: *Proc. Workshop on Programs as Data Objects*, Copenhagen, Lecture Notes in Computer Science **217** (Springer, Berlin, 1984).

[14] P.L. Wadler, Efficient compilation of pattern-matching, in: S.L. Peyton Jones, ed., *The Implementation of Functional Programming Languages* (Prentice Hall, Englewood Cliffs, NJ, 1987).

[15] P.L. Wadler, The concatenate vanishes, Note distributed to FP electronic mailing list, 1987.