Deforestation: Transforming programs to eliminate trees

Philip Wadler
University of Glasgow*

Intermediate lists—and, more generally, intermediate trees—are both the basis and the bane of a certain style of programming in lazy 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. For each list, time is required to allocate it, examine it, and deallocate it. Transforming the above to eliminate the intermediate lists gives

$$\begin{array}{l} h\ 0\ 1\ n \\ \textbf{where} \\ h\ a\ m\ n\ =\ \textbf{if}\ m > n \\ \qquad\qquad\quad \textbf{then}\ a \\ \qquad\qquad\quad \textbf{else}\ h\ (a + square\ m)\ (m + 1)\ n \end{array} \tag{2}$$

This program is more efficient because all operations on list cells have been eliminated.

This paper presents an algorithm that transforms programs to eliminate intermediate lists—and intermediate trees—called the Deforestation Algorithm. We characterise a form of function definition, *treeless form*, that uses no intermediate trees. 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 optimising compiler. Treeless form is easy to identify syntactically, and the transformation applies to any term (or sub-term) composed of treeless functions.

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

*This work was in part performed at Oxford University, under a research fellowship funded by ICL.

Author's address: Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland

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 added the LML compiler [Aug87,Joh87] by Kei Davis [Dav87]. 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 [Wad84,Wad85]. 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 [Wad85] 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.

The remainder of this paper is organised 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

We use a first-order language with the following grammar:

$$
\begin{array}{rcll}
t & ::= & v & \text{variable} \\
 & | & c\ t_1\ \ldots\ t_k & \text{constructor application} \\
 & | & f\ t_1\ \ldots\ t_k & \text{function application} \\
 & | & \textbf{case}\ t_0\ \textbf{of}\ p_1 : t_1 \mid \cdots \mid p_n : t_n & \text{case term} \\
 & & & \\
p & ::= & c\ v_1\ \ldots\ v_k & \text{pattern}
\end{array}
$$

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 Figure 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 [Aug85,Wad87a].

We assume that the language is typed using the Milner polymorphic typing system [Mil78,DM82,Han87], found in LML and Miranda[1] [Tur85], among others. Familiarity with this type system is assumed.

[1]Miranda is a trademark of Research Software Limited.

```
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)
```

Figure 1: Example definitions

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 *append* has arity 2. 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. We must extend this definition 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. We say one term is more efficient than another if, for every possible instantiation of the free variables, the first requires fewer 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, we have

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

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

Given a collection of function definitions F, we say that F is treeless if each right-hand side in F is treeless with respect to F. The definitions of *append* and *flip* in Figure 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

$$\mathit{flip}\ (\mathit{flip}\ zt)$$

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

$$\mathit{Branch}\ (\mathit{flip}\ yt)\ (\mathit{flip}\ xt)$$

where the trees returned by $(\mathit{flip}\ yt)$ and $(\mathit{flip}\ xt)$ 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 a left-hand side of an equation by the corresponding instance of the right-hand side [BD77]. Whenever we unfold a definition with a non-linear right-hand side, we risk 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, we would prefer our program to contain $\mathit{square}\ t$ rather than its unfolded equivalent $t \times t$. On the other hand, if we define $\mathit{square}\ x = \mathit{exp}\ (2 \times \mathit{log}\ x)$ then *square* is linear, and there is no harm in unfolding $\mathit{square}\ t$ to get $\mathit{exp}\ (2 \times \mathit{log}\ t)$. By insisting that treeless definitions are linear, we guarantee that we can unfold them without sacrificing efficiency.

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.)

We can now present our main result.

Deforestation Theorem. Every linear term, containing only occurrences of functions with treeless definitions, can be effectively transformed to an equivalent treeless

```
flatten0          : list (list α) → list α
flatten0 xss      = case xss of
                      Nil          : Nil
                      Cons xs xss  : append xs (flatten0 xss)

flatten1          : list (list α) → list α
flatten1 xss      = case xss of
                      Nil          : Nil
                      Cons xs xss  : flatten1' xs xss

flatten1'         : list α → list (list α) → list α
flatten1' xs xss  = case xs of
                      Nil          : flatten1 xss
                      Cons x xs    : Cons x (flatten1' xs xss)
```

Figure 2: A non-treeless and a treeless definition of *flatten*

term, without loss of efficiency.

We will name the algorithm that carries out the effective transformation the Deforestation Algorithm. 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.

For example, both *append* (*append xs ys*) *zs* and *flip* (*flip zt*) satisfy the hypothesis of the theorem. Applying the Deforestation Algorithm transforms these functions as shown in Figure 3. The transformation of *append* (*append xs ys*) *zs* is particularly noteworthy, since this term takes time $2m + n$ to compute, whereas the transformed version takes time $m + n$ to compute (where m is the length of *xs* and n is the length of *ys*). The transformation of this term 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 characterisation of treeless definitions and the hypothesis of the Deforestation Theorem are both 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.

append (*append xs ys*) *zs*

transforms to

$h_0\ xs\ ys\ zs$
where
$h_0\ xs\ ys\ zs$ = **case** *xs* **of**
Nil : $h_1\ ys\ zs$
Cons x xs : *Cons x* $(h_0\ xs\ ys\ zs)$
$h_1\ ys\ zs$ = **case** *ys* **of**
Nil : *zs*
Cons y ys : *Cons y* $(h_1\ ys\ zs)$

---

flip (*flip zt*)

transforms to

$h_0\ zt$
where
$h_0\ zt$ = **case** *zt* **of**
Leaf z : *Leaf z*
Branch xt yt : *Branch* $(h_0\ xt)$ $(h_0\ yt)$

Figure 3: Results of applying the Deforestation Algorithm

3 The Deforestation Algorithm

The heart of the Deforestation Algorithm is the seven rules shown in Figure 4. We write $T[\![t]\!]$ to denote the result of converting term t to treeless form. We must have 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.

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 $t'_1, \ldots, t'_n$. It is always possible to rename the bound variables so that this condition applies.)

There is one problem: the algorithm as given does not terminate! An example of applying rules

(1) $T[\![v]\!] = v$

(2) $T[\![c\ t_1\ \ldots\ t_k]\!] = c\ (T[\![t_1]\!])\ \ldots\ (T[\![t_k]\!])$

(3) $T[\![f\ t_1\ \ldots\ t_k]\!] = T[\![t[t_1/v_1, \ldots, t_k/v_k]]\!]$
where f is defined by $f\ v_1\ \ldots\ v_k = t$

(4) $T[\![\textbf{case}\ v\ \textbf{of}\ p'_1 : t'_1 \mid \cdots \mid p'_n : t'_n]\!]$
$= \textbf{case}\ v\ \textbf{of}\ p'_1 : T[\![t'_1]\!] \mid \cdots \mid p'_n : T[\![t'_n]\!]$

(5) $T[\![\textbf{case}\ c\ t_1\ \ldots\ t_k\ \textbf{of}\ p'_1 : t'_1 \mid \cdots \mid p'_n : t'_n]\!]$
$= T[\![t'_i[t_1/v_1, \ldots, t_k/v_k]]\!]$
where $p'_i = c\ v_1\ \ldots\ v_k$

(6) $T[\![\textbf{case}\ f\ t_1\ \ldots\ t_k\ \textbf{of}\ p'_1 : t'_1 \mid \cdots \mid p'_n : t'_n]\!]$
$= T[\![\textbf{case}\ t[t_1/v_1, \ldots, t_k/v_k]\ \textbf{of}\ p'_1 : t'_1 \mid \cdots \mid p'_n : t'_n]\!]$
where f is defined by $f\ v_1\ \ldots\ v_k = t$

(7) $T[\![\textbf{case}\ (\textbf{case}\ t_0\ \textbf{of}\ p_1 : t_1 \mid \cdots \mid p_m : t_m)\ \textbf{of}\ p'_1 : t'_1 \mid \cdots \mid p'_n : t'_n]\!]$
$= T[\![\textbf{case}\ t_0\ \textbf{of}$
$p_1\ :\ (\textbf{case}\ t_1\ \textbf{of}\ p'_1 : t'_1 \mid \cdots \mid p'_n : t'_n)$
$\ldots$
$p_m\ :\ (\textbf{case}\ t_m\ \textbf{of}\ p'_1 : t'_1 \mid \cdots \mid p'_n : t'_n)]\!]$

Figure 4: Transformation rules for the Deforestation Algorithm

(1–7) is shown in Figure 5. This shows transformation of the term

$$\mathit{flip}\ (\mathit{flip}\ zt)$$

After the steps shown, we finally reach the form

$$\begin{array}{lll} \textbf{case}\ zt\ \textbf{of} & & \\ \quad \mathit{Leaf}\ z & : & \mathit{Leaf}\ z \\ \quad \mathit{Branch}\ xt\ yt & : & \mathit{Branch}\ (T[\![\mathit{flip}\ (\mathit{flip}\ xt)]\!])\ (T[\![\mathit{flip}\ (\mathit{flip}\ yt)]\!]) \end{array} \qquad (*)$$

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

The key trick to avoid this infinite regress is to introduce appropriate *new* function definitions. For the example above, we introduce a function h_0 that satisfies the equation

$$h_0\ zt \;=\; T[\![\mathit{flip}\ (\mathit{flip}\ zt)]\!]$$

Now when the expansion of $T[\![\mathit{flip}\ (\mathit{flip}\ zt)]\!]$ reaches the form $(*)$ above, we can recognise that the two occurrences of $T[\![\ldots]\!]$ match the right-hand side of this equation, and replace them by the corresponding left-hand side, giving

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

This completes our task; we have translated the term $\mathit{flip}\ (\mathit{flip}\ zt)$ to the equivalent treeless term $h_0\ zt$, where h_0 has the treeless definition we have just derived.

When should we introduce new definitions? The answer is that *every* term of the form $T[\![\ldots]\!]$ encountered in the course of applying rules (1–7) is a potential right-hand side for a new definition. We keep a list of all such terms. Whenever we encounter a term for a second time, we create the appropriate function definition and replace each instance of the term by a corresponding call of the function. It is sufficient if the new term is a *renaming* of a previous term. For example, in the transformation above, $(\mathit{flip}\ (\mathit{flip}\ xt))$ was a renaming of $(\mathit{flip}\ (\mathit{flip}\ zt))$, and was replaced by the corresponding call $(h_0\ xt)$. We insist that the new term is 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 we get a term in treeless form, and that this term will itself be equivalent to t. Below we sketch a proof that whenever t satisfies the hypothesis of the Deforestation Theorem, then there is a bound on the size of the terms of the form $T[\![\ldots]\!]$ encountered while applying rules (1–7). Since the terms are bounded in size, 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 unfolding (rules (3) and (6)) never introduces 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 if t contains only occurrences of treeless functions, then there is a bound on the size of the terms encountered by the Deforestation Algorithm. Define the *nesting* of

```
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)⟧)
```

Figure 5: Deforestation of *flip* (*flip zt*)

a term t, written $N[\![t]\!]$, as follows:

$$\begin{array}{lcl} N[\![v]\!] & = & 0 \\ N[\![c\ t_1\ \ldots\ t_k]\!] & = & max(N[\![t_1]\!], \ldots, N[\![t_k]\!]) \\ N[\![f\ t_1\ \ldots\ t_k]\!] & = & 1 + max(N[\![t_1]\!], \ldots, N[\![t_k]\!]) \\ N[\![\textbf{case}\ t_0\ \textbf{of}\ p_1 : t_1 \mid \cdots \mid p_n : t_n]\!] & = & max(N[\![t_0]\!], N[\![t_1]\!], \ldots, N[\![t_k]\!]) \end{array}$$

It is easy to verify that the nesting of a treeless term is at most 1, and that unfolding a treeless definition never increases the nesting of a term. So the nesting of any term encountered by the Deforestation Algorithm is bounded by the nesting of the initial term; call this N. Further, let M be the maximum of the size of the initial term and the size of any right-hand side of a function definition referred to (directly or indirectly) from the initial term. Then the size of any term encountered by the Deforestation Algorithm can be shown to be bounded by MN. This guarantees that the Deforestation Algorithm terminates whenever the hypothesis of the Deforestation Theorem is satisfied, and so completes the proof of the theorem.

Even if the given term does not satisfy the hypothesis of the Deforestation Theorem, the algorithm may still terminate, and when it does it will return an equivalent treeless term. For example, applying the algorithm to the non-treeless definition of $flatten_0$ in Figure 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, which we will henceforth call *pure* treeless form, is quite restrictive. Consider the definition

$$\begin{array}{lcl} upto & : & int \to int \to list\ int \\ upto\ m\ n & = & \textbf{case}\ (m > n)\ \textbf{of} \\ & & \quad True\ :\ Nil \\ & & \quad False\ :\ Cons\ m\ (upto\ (m+1)\ n) \end{array}$$

For example, *upto* 1 4 returns $[1,2,3,4]$. Here we write $t_1 + t_2$ as an abbreviation for $(+)\ t_1\ t_2$, where $(+)$ is considered a function name; and similarly for other infix operators.

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*, we will divide all terms 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 we will call the mark $\oplus$ or $\ominus$ 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, we will blaze all terms of type $(list\ \alpha)$ or $(tree\ \alpha)$ with $\oplus$, and all terms of type *int* or *bool* with $\ominus$. If t stands for an arbitrary term, we will write $t^{\oplus}$ to indicate that t is of a type blazed $\oplus$, and $t^{\ominus}$ to indicate that t is of a type blazed $\ominus$.

In the definition of pure treeless form, the places where intermediate values can 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 can 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:

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

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 Figure 6 are treeless.

The Deforestation Theorem carries over virtually unchanged:

Blazed Deforestation Theorem. Every term linear in variables blazed $\oplus$, con-

```
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)
```

Figure 6: More example definitions

taining only occurrences of functions with blazed treeless definitions, can be effectively transformed to an equivalent blazed treeless term, without loss of efficiency.

Two examples of applying the Blazed Deforestation Algorithm are shown in Figure 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. We will only introduce **let** terms through extraction, so the bound variable will always be blazed $\ominus$.

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

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

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

transforms to

$$\begin{array}{l}h_0\ 0\ 1\ n\\ \textbf{where}\\ h_0\ u_0\ u_1\ n\ \ =\ \ \textbf{case}\ \ (u_0 > n)\ \ \textbf{of}\\ \qquad\qquad True\ \ :\ \ a\\ \qquad\qquad False\ \ :\ \ h_0\ (u_0 + square\ u_1)\ (u_1 + 1)\ n\end{array}$$

$sumtr\ (squaretr\ xt)$

transforms to

$$\begin{array}{l}h_0\ xt\\ \textbf{where}\\ h_0\ xt\ \ =\ \ \textbf{case}\ \ xt\ \ \textbf{of}\\ \qquad\qquad Leaf\ x\ \ :\ \ square\ x\\ \qquad\qquad Branch\ xt\ yt\ \ :\ \ h_0\ xt + h_0\ yt\end{array}$$

Figure 7: Results of applying the Blazed Deforestation Algorithm

yields the term

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

The inner term here is a renaming of the inner term of the previous expression, and will cause the appropriate new function to be defined:

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

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.

We must also add to the definition of $T[\![t]\!]$ in Figure 4 the four additional rules in Figure 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 in any of $p'_1, \ldots, p'_m$. It is always possible to rename the bound variables so that this condition applies.)

After the transformation is complete, all terms of the form

$$\textbf{let } v^{\ominus} = tt_0^{\ominus} \textbf{ in } tt_1$$

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

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

(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) $T[\![\textbf{case } (\textbf{let } v^{\ominus} = t_0^{\ominus} \textbf{ in } t_1) \textbf{ of } p'_1 : t'_1 \mid \cdots \mid p'_n : t'_n]\!]$
$= \textbf{let } v^{\ominus} = T[\![t_0^{\ominus}]\!] \textbf{ in } T[\![\textbf{case } t_1 \textbf{ of } p'_1 : t'_1 \mid \cdots \mid p'_n : t'_n]\!]$

Figure 8: Additional rules for the Blazed Deforestation Algorithm

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.

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

$$t \textbf{ where } d_1; \ldots; d_n$$

where t is a term and $d_1, \ldots, d_n$ are function definitions. This can be translated back into our equation language in a straightforward manner, by use of a technique called *lambda lifting* [Joh87,Pey87]. 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

$$f\ v_1\ \ldots\ v_k \ \hat{=}\ t$$

That is, they are like ordinary definitions, but we write $\hat{=}$ instead of $=$. The term t may now contain variables in place of function names, and applications are no longer restricted by arity. The same Milner polymorphic type system is used. The formal parameters $v_1, \ldots, v_n$ may now have a ground type, like *int* or $(list\ \alpha)$, or a function type, like $(int \to int)$, or even a higher-order type, like $((int \to int) \to int)$. The only restriction is that *higher-order macros cannot be recursive.*

The lack of recursion, combined with the 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. Doesn't it rule out our favourite higher-order functions, such as *map* and *fold*? As it turns out, it doesn't: we get the recursion back by using the **where** facility defined above. Definitions of *map* and *fold* are given in Figure 9; recursion is limited to the first-order functions g and h.

Given the definitions in Figure 9 we can write terms such as

$$sum\ (map\ square\ (upto\ 1\ n))$$
$$map\ sum\ (map\ (map\ square)\ xss)$$
$$(map\ square \circ map\ cube)\ xs$$
$$map\ (square \circ cube)\ xs$$

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. For instance, using this mechanism it is not possible to define or manipulate a list of functions, as one could in a true higher-order language.

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.

```
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 x)

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)
```

Figure 9: Example higher-order definitions

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 key steps in the LML compiler involve transformation techniques [Aug87,Joh87]. 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 recognised 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 characterise what sort of expressions will be optimised and what sort of optimisations will be performed.

Further work is desirable in two directions.

First, treeless form may be generalised. One possible generalisation 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 generalisation, it would. Related ideas are discussed in [Wad87b].

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

Acknowledgements. 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.

comments on this paper.

References

[Aug85] L. Augustsson, Compiling pattern matching. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, Nancy, France, September 1985. LNCS 201, Springer-Verlag, 1985.

[Aug87] L. Augustsson, Compiling lazy functional languages, Part II. Ph.D. dissertation, Department of Computer Science, Chalmers Tekniska Högskola, Göteborg, Sweden, 1987.

[BD77] R. M. Burstall and J. Darlington, A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[Dav87] M. K. Davis, Deforestation: Transformation of functional programs to eliminate intermediate trees. M.Sc. dissertation, Programming Research Group, Oxford University, September 1987.

[DM82] L. Damas and R. Milner, Principal type schemes for functional programs. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, January 1982.

[Joh87] T. Johnsson, Compiling lazy functional languages. Ph.D. dissertation, Department of Computer Science, Chalmers Tekniska Högskola, Göteborg, Sweden, 1987.

[Han87] P. Hancock, Polymorphic type-checking. In [Pey87].

[Mil78] R. Milner, A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Pey87] S. L. Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice Hall, 1987.

[Tur85] D. A. Turner, Miranda: A non-strict functional language with polymorphic types. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, Nancy, France, September 1985. LNCS 201, Springer-Verlag, 1985.

[Wad84] P. L. Wadler, Listlessness is better than laziness: Lazy evaluation and garbage collection at compile-time. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, Austin, Texas, August 1984.

[Wad85] P. L. Wadler, Listlessness is better than laziness II: Composing listless functions. In *Proceedings of the Workshop on Programs as Data Objects*, Copenhagen, October 1985. LNCS 217, Springer-Verlag, 1985.

[Wad87a] P. L. Wadler, Efficient compilation of pattern-matching. In [Pey87].

[Wad87b] P. L. Wadler, The concatenate vanishes. Note distributed to FP electronic mailing list, December 1987.