

# Higher Order Deforestation

G.W.Hamilton

Technical Report TR95-07

July 1995

Department of Computer Science

University of Keele

Keele

Staffordshire

ST5 5BG

U.K.

TEL : 01782 621111

FAX : 01782 713082

## Abstract

Intermediate data structures are widely used in functional programs. Programs which use these intermediate structures are usually a lot easier to understand, but they result in loss of efficiency at run-time. In order to reduce these run-time costs, a transformation algorithm called *deforestation* was proposed by Wadler which could eliminate intermediate structures. However, this transformation algorithm was formulated only for first order functional programs. In this paper, it is shown how the original deforestation algorithm can be extended to deal with higher order functional programs. This extended algorithm is guaranteed to terminate only for expressions in which all functions are in a *treeless* form. It is shown how all function definitions can be generalised to this form so that the algorithm can be made to terminate for all programs. It is therefore argued that this algorithm is suitable for inclusion in an optimising compiler.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Language</b>	<b>1</b>
<b>3</b>	<b>Higher Order Treeless Form</b>	<b>2</b>
<b>4</b>	<b>The Higher Order Deforestation Algorithm</b>	<b>4</b>
<b>5</b>	<b>The Higher Order Deforestation Theorem</b>	<b>7</b>
<b>6</b>	<b>Related Work</b>	<b>10</b>
<b>7</b>	<b>Conclusion</b>	<b>11</b>

# List of Figures

1	Language Grammar . . . . .	1
2	Example Program . . . . .	2
3	Higher Order Treeless Form . . . . .	3
4	Result of Applying Higher Order Deforestation Algorithm . . . . .	4
5	Transformation Rules for Higher Order Deforestation . . . . .	5
6	Modified Transformation Rules for Deforestation . . . . .	6
7	Result of Applying Higher Order Deforestation Algorithm . . . . .	7
8	Size of Higher Order Expressions . . . . .	9
9	Grammar of Expressions Encountered During Higher Order Deforestation . . . . .	9

# 1 Introduction

The use of intermediate structures, lazy evaluation and higher order functions in functional programming facilitates a more elegant and readable style of programming (see [6]), but it often results in inefficient programs. One solution to this problem is to transform these programs to more efficient equivalent programs. A transformation algorithm called *deforestation* was proposed in [13] which can eliminate intermediate structures from first order functional programs. This algorithm was extended to allow a restricted higher order facility by treating higher order functions as macros. This mechanism does not cover all uses of higher order functions.

In this paper, it is shown how the deforestation algorithm described in [13] can be extended to handle higher order functions directly. As in [13], a *treeless* form of function definition which creates no intermediate trees is defined. Unfortunately, it is found that most useful higher order functions are not in this treeless form. It is therefore shown how function definitions can be generalised so that they are in this form. The higher order deforestation algorithm, which can transform any expression in which all functions are in treeless form into an equivalent treeless expression, is then defined, and it is proved that it will always terminate if all functions are in treeless form.

The remainder of this paper is organised as follows. In Section 2, the higher order language on which the described transformations are to be performed is introduced. In Section 3, higher order treeless form is defined, and it is shown how function definitions can be generalised so that they are in this form. In Section 4, the higher order deforestation algorithm is described and, in Section 5, it is proved that it will always terminate if all functions are in treeless form. In Section 6, related work is discussed, and Section 7 concludes.

## 2 Language

The language for which the described transformations are to be performed is a simple higher order functional language.

### Definition 2.1 (Language Grammar)

The grammar of the language is as given in Figure 1.

□

$prog$	$::=$	$e_0$	
		<b>where</b>	
		$f_1 = e_1$	program
		$\vdots$	
		$f_n = e_n$	
$e$	$::=$	$v$	variable
		$c e_1 \dots e_n$	constructor application
		$f$	function variable
		$\lambda v. e_0$	lambda expression
		<b>case</b> $e_0$ <b>of</b> $p_1 : e_1 \mid \dots \mid p_k : e_k$	case expression
		$e_0 e_1$	function application
$p$	$::=$	$c v_1 \dots v_n$	pattern

Figure 1: Language Grammar

An example program in the language is shown in Figure 2.

```

fold (+) 0 (map square (until (> n) (repeat (+ 1) 1)))
where
fold    = λf.λa.λxs.case xs of
          Nil      : a
          Cons x xs : fold f (f a x) xs

map     = λf.λxs.case xs of
          Nil      : Nil
          Cons x xs : Cons (f x) (map f xs)

until   = λp.λxs.case xs of
          Nil      : Nil
          Cons x xs : case (p x) of
                        True   : Nil
                        False  : Cons x (until p xs)

repeat = λf.λx.Cons x (repeat f (f x))

```

Figure 2: Example Program

This program calculates the sum of the squares of the numbers from 1 to  $n$ . The function *repeat* is used to create a list of integers starting with 1, with each subsequent number increasing by 1. The function *until* is used to select the first  $n$  of these. The function *map* is used to square each of these numbers, and the function *fold* is used to add these together.

Programs in the language consist of an expression to evaluate and a set of function definitions. It is assumed that the compiler for the language implements the full laziness technique described in [5]. Nested function definitions are not allowed in the language. Programs involving nested definitions can be transformed into this restricted form of program using a technique called *lambda lifting* [7].

For the purposes of this paper, constants (e.g. numbers) and basic functions (+, -, <, etc.) can be considered to be variables. Each constructor has a fixed arity (for example, *Nil* has arity 0, and *Cons* has arity 2) and each constructor application must be saturated. It is assumed that the language is typed using the Milner polymorphic typing system [9, 3].

Within **case** expressions of the given form,  $e_0$  is called the *selector*, and  $p_1 : e_1 \dots p_k : e_k$  are called the *branches*. Branches can be separated by either the | or the newline character. The patterns in **case** expressions may not be nested. Methods to transform **case** expressions with nested patterns to ones without nested patterns are described in [1] and [12].

As in [13], an expression other than a **case** expression is defined to be *linear* with respect to a variable if the number of occurrences of the variable within the expression is not more than one. A **case** expression is defined to be linear with respect to a variable if the total number of occurrences of the variable within the selector and any branch of the expression is not more than one. For example, the function *fold* given in Figure 2 is linear with respect to the variable  $a$ , but it is not linear with respect to the variable  $f$ .

### 3 Higher Order Treeless Form

In this section, a higher order treeless form is expression is defined which is similar to the first order blazed treeless form defined in [13]. In the first order blazed treeless form defined in [13], expressions are *blazed*<sup>1</sup> according to whether or not they can be eliminated by the deforestation algorithm. Expressions blazed  $\oplus$  can be eliminated, but expressions blazed  $\ominus$  cannot. This blazing was performed according to

---

<sup>1</sup> In forestry, blazing is the operation of marking a tree by making a cut in its bark.

the type of the expressions; expressions of structured type are blazed  $\oplus$  and expressions of basic type are blazed  $\ominus$ .

In higher order treeless form, expressions are blazed as follows. Function arguments and **case** selectors which are not variables are blazed  $\ominus$ ; these are intermediate structures which will be transformed separately and will not be removed by the higher order deforestation algorithm. No other expressions are blazed; these are equivalent to expressions which are blazed  $\oplus$  in [13]<sup>2</sup>.

In higher order treeless form, variables can also be blazed at their binding occurrence<sup>3</sup>. All non-linear variables are blazed in this way. Expressions cannot be substituted for blazed variables, and will be transformed separately by the higher order deforestation algorithm. This ensures that expressions which are expensive to compute will not be duplicated<sup>4</sup>. For example, consider a function call *square*  $e$  where *square* is a non-linear function defined as  $\lambda x.x * x$ . If  $e$  is an expression which is expensive to compute, then the unfolded expression  $e * e$  will be less efficient than the original expression *square*  $e$ . This situation is avoided by blazing the bound variable  $x$  to give the function definition  $\lambda x^\ominus.x * x$ . The function argument will be transformed separately in any applications, and will not cause any duplication of work.

Higher order treeless form is defined as follows.

**Definition 3.1 (Higher Order Treeless Form)** *An expression is in higher order treeless form if all function arguments and case selectors are blazed, and each non-linear variable is blazed at its binding occurrence.*

□

Expressions in higher order treeless form must therefore satisfy the grammar given in Figure 3 where, in addition, each non-linear variable is blazed at its binding occurrence.

$te ::= v$	$c\ e_1 \dots e_n$	where $e_i \in te\ \forall i \in \{1 \dots n\}$
$f$	$\lambda v.e$	where $f$ is defined by $f = e$ and $e \in te$ where $e \in te$
$\mathbf{case}\ e_0\ \mathbf{of}\ p_1 : e_1\   \dots\   p_k : e_k$	$e_0\ e_1$	where $e_0 \in te'$ and $e_i \in te\ \forall i \in \{1 \dots k\}$ where $e_0 \in te$ and $e_1 \in te'$
$te' ::= v$	$e^\ominus$	where $e \in te$

Figure 3: Higher Order Treeless Form

Unfortunately, a lot of useful higher order functions are not in higher order treeless form. For example, none of the function definitions given in Figure 2 are in this form. However, function definitions can be generalised so that they are in higher order treeless form as follows. An expression is not in higher order treeless form if any **case** selector or function argument is not a variable or a blazed expression. All **case** selectors and function arguments which are not variables are therefore blazed. An expression is also not in higher order treeless form if it is non-linear in any variable which is not blazed at its binding occurrence. All non-linear variables are therefore blazed at their binding occurrence. The program given in Figure 2 would therefore be blazed as shown in Figure 4.

<sup>2</sup> For the remainder of this paper if an expression is said to be blazed, it is assumed that it is annotated with  $\ominus$ , as  $\ominus$  is the only blazing annotation which is used.

<sup>3</sup> This is the occurrence of a variable as the bound variable in a lambda abstraction, or within the pattern of a case expression.

<sup>4</sup> This relies upon the fact that the language is implemented using the full laziness technique described in [5], since it is possible to duplicate references to a partially applied function which contains expressions that will be re-evaluated when it is fully applied.

```

fold (+) 0 (map square (until (> n) (repeat (+ 1) 1)))
where
fold    =  λf⊖.λa.λxs.case xs of
           Nil      : a
           Cons x xs : fold f (f a x)⊖ xs

map     =  λf⊖.λxs.case xs of
           Nil      : Nil
           Cons x xs : Cons (f x) (map f xs)

until   =  λp⊖.λxs.case xs of
           Nil      : Nil
           Cons x⊖ xs : case (p x)⊖ of
                       True  : Nil
                       False : Cons x (until p xs)

repeat  =  λf⊖.λx⊖.Cons x (repeat f (f x)⊖)

transform to:

g 0 1 n
where
g    =  λa.λm.λn.case (m > n)⊖ of
       True  : a
       False : g (a + square m)⊖ (m + 1)⊖ n

```

Figure 4: Result of Applying Higher Order Deforestation Algorithm

## 4 The Higher Order Deforestation Algorithm

The higher order deforestation algorithm is a set of transformation rules which attempt to convert a given expression into a higher order treeless equivalent. The transformation rules for the given language are shown in Figure 5. These rules cover all possible kinds of expression. Rule (1) deals with blazed expressions. The remaining rules deal with the five other kinds of expression (variable, constructor, function, lambda and **case**). These are all formulated as applications of  $n$  arguments<sup>5</sup>. All six possibilities for the selector of a **case** expression are considered in rules (6) - (11).

Rules (5) - (8) and (10) - (11) are valid only if there is no name clash between the free and bound variables of the expression being transformed. It is always possible to rename the bound variables of the expression so that this condition applies.

As is the case for the first order deforestation algorithm described in [13], the higher order algorithm as given will not necessarily terminate. Termination is achieved only through the introduction of appropriate new function definitions.

Any infinite sequence of transformation steps must involve the unfolding of a function definition<sup>6</sup>. A new function definition is therefore introduced before the application of rules (4) and (9). The right hand sides of these function definitions are the expressions which were about to be transformed. When an expression is encountered later in the transformation which matches the right hand side of one of these function definitions (modulo renaming of variables), it is replaced by an appropriate call of the corresponding function. Transformation rules (4) and (9) must therefore be changed as shown in Figure

<sup>5</sup> A simple expression can be regarded as an application with no arguments (i.e.  $n = 0$ ).

<sup>6</sup> Expressions such as  $(\lambda x.x x)$   $(\lambda x.x x)$  are not allowed in a well typed language, so cannot be involved in an infinite sequence of transformation steps. Such expressions would not result in an infinite sequence of transformation steps anyway because of the linearity constraint.

$$\begin{aligned}
(1) \quad \mathcal{T}[[e^\ominus]] &= (\mathcal{T}[[e]])^\ominus \\
(2) \quad \mathcal{T}[[v \ e_1 \dots e_n]] &= v \ (\mathcal{T}[[e_1]])^\ominus \dots (\mathcal{T}[[e_n]])^\ominus \\
(3) \quad \mathcal{T}[[c \ e_1 \dots e_n]] &= c \ (\mathcal{T}[[e_1]])^\ominus \dots (\mathcal{T}[[e_n]])^\ominus \\
(4) \quad \mathcal{T}[[f \ e_1 \dots e_n]] &= \mathcal{T}[[e \ e_1 \dots e_n]] \\
&\quad \text{where } f \text{ is defined by } f = e \\
(5a) \quad \mathcal{T}[[\lambda v. e_0 \ e_1^\ominus \dots e_n]] &= (\lambda v. \mathcal{T}[[e_0 \ e_2 \dots e_n]]) \ (\mathcal{T}[[e_1]])^\ominus \\
(5b) \quad \mathcal{T}[[\lambda v^\ominus. e_0 \ e_1 \dots e_n]] &= (\lambda v^\ominus. \mathcal{T}[[e_0 \ e_2 \dots e_n]]) \ (\mathcal{T}[[e_1]])^\ominus \\
(5c) \quad \mathcal{T}[[\lambda v. e_0 \ e_1 \dots e_n]] &= \lambda v. \mathcal{T}[[e_0]], \quad \text{if } n = 0 \\
&= \mathcal{T}[[e_0[e_1/v] \ e_2 \dots e_n]], \quad \text{otherwise} \\
(6) \quad \mathcal{T}[[\text{case } e_0^\ominus \ \text{of } p'_1 : e'_1 \mid \dots \mid p'_k : e'_k \ e''_1 \dots e''_n]] &= \text{case } (\mathcal{T}[[e_0]])^\ominus \ \text{of } p'_1 : \mathcal{T}[[e'_1 \ e'_1 \dots e''_n]] \mid \dots \mid p'_k : \mathcal{T}[[e'_k \ e'_1 \dots e''_n]] \\
(7) \quad \mathcal{T}[[\text{case } (v \ e_1 \dots e_m) \ \text{of } p'_1 : e'_1 \mid \dots \mid p'_k : e'_k \ e''_1 \dots e''_n]] &= \mathcal{T}[[\text{case } (v \ e_1 \dots e_m)^\ominus \ \text{of } p'_1 : e'_1 \mid \dots \mid p'_k : e'_k \ e''_1 \dots e''_n]] \\
(8) \quad \mathcal{T}[[\text{case } (c \ e_1 \dots e_m) \ \text{of } p'_1 : e'_1 \mid \dots \mid p'_k : e'_k \ e''_1 \dots e''_n]] &= \mathcal{T}[[\lambda v_1 \dots v_m. e'_i \ e''_1 \dots e''_n \ e_1 \dots e_m]] \\
&\quad \text{where } p'_i = c \ v_1 \dots v_m \\
(9) \quad \mathcal{T}[[\text{case } (f \ e_1 \dots e_m) \ \text{of } p'_1 : e'_1 \mid \dots \mid p'_k : e'_k \ e''_1 \dots e''_n]] &= \mathcal{T}[[\text{case } (e \ e_1 \dots e_m) \ \text{of } p'_1 : e'_1 \mid \dots \mid p'_k : e'_k \ e''_1 \dots e''_n]] \\
&\quad \text{where } f \text{ is defined by } f = e \\
(10a) \quad \mathcal{T}[[\text{case } ((\lambda v. e_0) \ e_1^\ominus \dots e_m) \ \text{of } p'_1 : e'_1 \mid \dots \mid p'_k : e'_k \ e''_1 \dots e''_n]] &= (\lambda v. \mathcal{T}[[\text{case } (e_0 \ e_2 \dots e_m) \ \text{of } p'_1 : e'_1 \mid \dots \mid p'_k : e'_k \ e''_1 \dots e''_n]]) \ (\mathcal{T}[[e_1]])^\ominus \\
(10b) \quad \mathcal{T}[[\text{case } ((\lambda v^\ominus. e_0) \ e_1 \dots e_m) \ \text{of } p'_1 : e'_1 \mid \dots \mid p'_k : e'_k \ e''_1 \dots e''_n]] &= (\lambda v^\ominus. \mathcal{T}[[\text{case } (e_0 \ e_2 \dots e_m) \ \text{of } p'_1 : e'_1 \mid \dots \mid p'_k : e'_k \ e''_1 \dots e''_n]]) \ (\mathcal{T}[[e_1]])^\ominus \\
(10c) \quad \mathcal{T}[[\text{case } ((\lambda v. e_0) \ e_1 \dots e_m) \ \text{of } p'_1 : e'_1 \mid \dots \mid p'_k : e'_k \ e''_1 \dots e''_n]] &= \mathcal{T}[[\text{case } (e_0[e_1/v] \ e_2 \dots e_m) \ \text{of } p'_1 : e'_1 \mid \dots \mid p'_k : e'_k \ e''_1 \dots e''_n]] \\
(11) \quad \mathcal{T}[[\text{case } ((\text{case } e_0 \ \text{of } p_1 : e_1 \mid \dots \mid p_j : e_j) \ e'_1 \dots e'_m) \ \text{of } p''_1 : e''_1 \mid \dots \mid p''_k : e''_k \ e'''_1 \dots e'''_n]] &= \mathcal{T}[[\text{case } e_0 \ \text{of} \\
&\quad p_1 \ : \ \text{case } (e_1 \ e'_1 \dots e'_m) \ \text{of } p''_1 : e''_1 \mid \dots \mid p''_k : e''_k \\
&\quad \vdots \\
&\quad p_j \ : \ \text{case } (e_j \ e'_1 \dots e'_m) \ \text{of } p''_1 : e''_1 \mid \dots \mid p''_k : e''_k \ e'''_1 \dots e'''_n]]
\end{aligned}$$

Figure 5: Transformation Rules for Higher Order Deforestation

$$\begin{aligned}
(4) \quad & \mathcal{T}[[f \ e_1 \dots e_n]] \phi \\
&= f' \ v_1 \dots v_j, \quad \text{if } (f' = \lambda v'_1 \dots v'_j . e) \in \phi \text{ and } (f \ e_1 \dots e_n) = e[v_1/v'_1, \dots, v_j/v'_j] \\
&\quad \text{where} \\
&\quad v_1 \dots v_j \text{ are the free variables in } (f \ e_1 \dots e_n) \\
&= f'' v_1 \dots v_j, \quad \text{otherwise} \\
&\quad \text{where} \\
&\quad f \text{ is defined by } f = e \\
&\quad f'' = \lambda v_1 \dots v_j . (\mathcal{T}[[e \ e_1 \dots e_n]] \phi') \\
&\quad \phi' = \phi \cup \{f'' = \lambda v_1 \dots v_j . f \ e_1 \dots e_n\} \\
&\quad v_1 \dots v_j \text{ are the free variables in } (f \ e_1 \dots e_n) \\
(9) \quad & \mathcal{T}[[\text{case } (f \ e_1 \dots e_m) \ \text{of } p'_1 : e'_1 \mid \dots \mid p'_k : e'_k \ e''_1 \dots e''_n]] \phi \\
&= f' \ v_1 \dots v_j, \quad \text{if } (f' = \lambda v'_1 \dots v'_j . e) \in \phi \text{ and} \\
&\quad ((\text{case } (f \ e_1 \dots e_m) \ \text{of } p'_1 : e'_1 \mid \dots \mid p'_k : e'_k \ e''_1 \dots e''_n) = e[v_1/v'_1, \dots, v_j/v'_j]) \\
&\quad \text{where} \\
&\quad v_1 \dots v_j \text{ are the free variables in } ((\text{case } (f \ e_1 \dots e_m) \ \text{of } p'_1 : e'_1 \mid \dots \mid p'_k : e'_k \ e''_1 \dots e''_n)) \\
&= f'' v_1 \dots v_j, \quad \text{otherwise} \\
&\quad \text{where} \\
&\quad f \text{ is defined by } f = e \\
&\quad f'' = \lambda v_1 \dots v_j . (\mathcal{T}[[\text{case } (e \ e_1 \dots e_m) \ \text{of } p'_1 : e'_1 \mid \dots \mid p'_k : e'_k \ e''_1 \dots e''_n]] \phi') \\
&\quad \phi' = \phi \cup \{f'' = \lambda v_1 \dots v_j . (\text{case } (f \ e_1 \dots e_m) \ \text{of } p'_1 : e'_1 \mid \dots \mid p'_k : e'_k \ e''_1 \dots e''_n)\} \\
&\quad v_1 \dots v_j \text{ are the free variables in } ((\text{case } (f \ e_1 \dots e_m) \ \text{of } p'_1 : e'_1 \mid \dots \mid p'_k : e'_k \ e''_1 \dots e''_n))
\end{aligned}$$

Figure 6: Modified Transformation Rules for Deforestation

6 to make this more explicit. In these rules, the additional parameter  $\phi$  contains the set of function definitions which have been created so far during the transformation. This additional parameter must also be passed to all other transformation rules.

The result of transforming the program in Figure 2 is shown in Figure 4. It can be seen that all the intermediate lists in the program have been eliminated. Another example of applying the higher order deforestation algorithm is shown in Figure 7.

The program being transformed can be used to determine whether a complex number  $k$ , represented by a pair, is a member of the Mandelbrot set. This is the case if the iterative formula  $z := z^2 + k$  converges with an initial value of  $(0,0)$  for  $z$ . It is impossible to find all the points for which the iteration converges, but it is possible to find an approximation to the Mandelbrot set. The sequence of iterations will diverge if the size of the complex number exceeds 2. Any point for which the iteration has not diverged after a fixed number of iterations can be assumed to lie within the set.

In the program given in Figure 7, the function *next* is used to calculate the value of the next iteration of the formula from the previous one. The function *repeat* is used to produce an infinite list of the values given by the iterations of the formula. The function *until* is used to take the values in this list until it is found that they will diverge. The function *morethan* determines whether more than  $n$  iterations are required before it can be determined that the sequence diverges. If this is the case, then the point  $k$  is assumed to be in the Mandelbrot set. It is assumed that the functions  $+$ , *square* and *size* are defined on complex numbers and are built-in functions of the language. The result of transforming this program is also shown in Figure 7. Again, it can be seen that all the intermediate lists in the program have been eliminated.

```

morethan n (until (diverges) (repeat (next k) (0,0)))
where
morethan  =  λn.λxs.case n of
              Zero    : True
              Succ n  : case xs of
                          Nil      : False
                          Cons x xs : morethan n xs

until      =  λp.λxs.case xs of
              Nil      : Nil
              Cons x xs : case (p x)⊖ of
                          True     : Nil
                          False    : Cons x (until p xs)

diverges   =  λz.(size z)⊖ > 2

repeat     =  λf.λx.Cons x (repeat f (f x)⊖)

next       =  λk.λz.k + (square z)⊖

```

transforms to:

```

f n k (0,0)
where
f          =  λn.λk.λz.case n of
              Zero    : True
              Succ n  : case ((size z)⊖ > 2)⊖ of
                          True     : False
                          False    : f n k (k + (square z)⊖)⊖

```

Figure 7: Result of Applying Higher Order Deforestation Algorithm

## 5 The Higher Order Deforestation Theorem

The deforestation theorem given in [13] can now be extended to the higher order deforestation theorem as follows.

**Theorem 5.1 (Higher Order Deforestation Theorem)** *Every expression in which each non-linear variable is blazed at its binding occurrence, and which contains only occurrences of functions with higher order treeless definitions can be transformed to an equivalent higher order treeless expression without loss of efficiency.*

□

### Proof

The higher order deforestation theorem can be proved by showing the following four lemmata, which together demonstrate the validity of the theorem.

**Lemma 5.2 (on equivalence)** *Every expression will be transformed to an equivalent expression if the higher order deforestation algorithm terminates.*

□

**Proof**

The proof of this lemma is similar to the proof of correctness of higher order deforestation given in [11].

□

**Lemma 5.3 (on higher order treeless form)** *Every expression in which each non-linear variable is blazed at its binding occurrence and which contains only occurrences of functions with higher order treeless definitions will be transformed to a higher order treeless expression if the higher order deforestation algorithm terminates.*

□

**Proof**

In order to prove that every expression will be transformed to a higher order treeless expression, it is shown that the result of each transformation rule is in higher order treeless form. The proof is by recursion induction over the transformation rules  $\mathcal{T}$ .

□

**Lemma 5.4 (on efficiency)** *Every expression in which each non-linear variable is blazed at its binding occurrence and which contains only occurrences of functions with higher order treeless functions will be transformed without loss of efficiency if the higher order deforestation algorithm terminates.*

□

**Proof**

The proof of this lemma is similar to the proof of efficiency of higher order deforestation given in [11].

□

**Lemma 5.5 (on termination)** *The higher order deforestation algorithm will terminate for every expression which contains only occurrences of functions with higher order treeless definitions.*

□

**Proof**

In order to be able to prove that the higher order deforestation algorithm always terminates, it is sufficient to show that there is a bound on the size of expressions encountered during transformation. If there is such a bound, then there will be a finite number of expressions encountered (modulo renaming of variables), and a renaming of a previous expression must eventually be encountered. The algorithm will therefore be guaranteed to terminate.

In order to prove that there is a bound on the size of expressions encountered during transformation, it is shown that all expressions which are encountered must satisfy a particular grammatical form. It is then shown that there is a bound on the size of the expressions described by this grammar. First of all, it must be defined what is meant by the size of an expression. This definition corresponds to the definition of the *depth* of an expression given in [13].

**Definition 5.6 (Size of Expressions)** *The size of an expression is defined as shown in Figure 8.*

□

**Definition 5.7 (Grammar of Expressions Encountered During Higher Order Deforestation)** *The grammar of expressions encountered during higher order deforestation is given by  $dg^s(s, n)$  for a suitable value of  $n$ , where  $s$  is the maximum size of all function definitions accessible within the expression and the grammar  $dg^s(x, y)$  is defined as shown in Figure 9.*

□

$\mathcal{S}[[v]]$	$=$	$0$
$\mathcal{S}[[c\ e_1 \dots e_n]]$	$=$	$1 + \max(\mathcal{S}[[e_1]], \dots, \mathcal{S}[[e_n]])$
$\mathcal{S}[[f]]$	$=$	$0$
$\mathcal{S}[[\lambda v.e]]$	$=$	$1 + \mathcal{S}[[e]]$
$\mathcal{S}[[\text{case } e_0 \text{ of } p_1 : e_1 \mid \dots \mid p_k : e_k]]$	$=$	$\max((1 + \mathcal{S}[[e_0]]), \mathcal{S}[[e_1]], \dots, \mathcal{S}[[e_k]])$
$\mathcal{S}[[e_1\ e_2]]$	$=$	$\max(\mathcal{S}[[e_1]], (1 + \mathcal{S}[[e_2]])$

Figure 8: Size of Higher Order Expressions

$dg^s(x, y)$	$::=$	$v$	if $x \geq 0$ and $y \geq 0$
		$c\ dg_1^s(x-1, y) \dots dg_n^s(x-1, y)$	if $x > 0$ and $y \geq 0$
		$f$ where $f = e$ and $e \in dg^s(s, 0)$	if $x \geq 0$ and $y \geq 0$
		$\lambda v. dg^s(x-1, y)$	if $x > 0$ and $y \geq 0$
		$\text{case } arg^s(x-1, y) \text{ of } p_1 : dg_1^s(x, y) \mid \dots \mid p_k : dg_k^s(x, y)$	if $x > 0$ and $y \geq 0$
		$dg^s(x, y)\ arg^s(x-1, y)$	if $x > 0$ and $y \geq 0$
		$dg^s(s, y-1)$	if $x \geq 0$ and $y > 0$
$arg^s(x, y)$	$::=$	$(dg^s(x, y))^\ominus$	if $x \geq 0$ and $y \geq 0$
		$dg^s(0, y)$	if $x \geq 0$ and $y \geq 0$

Figure 9: Grammar of Expressions Encountered During Higher Order Deforestation

In the definition of the grammar of expressions  $dg^s(x, y)$ ,  $y$  corresponds to the *order* of the expression (as described in [13]). All higher order treeless function definitions satisfy the grammar  $dg^s(s, 0)$ , since  $s$  is the maximum size of function definitions.

Lemma 5.5 can now be proved by showing the following two lemmata.

**Lemma 5.8 (on grammar of expressions encountered during higher order deforestation)** *All expressions encountered by the higher order deforestation algorithm are described by the grammar  $dg^s(s, n)$  if the original expression to be transformed is also described by the grammar  $dg^s(s, n)$  for a suitable value of  $n$ , where  $s$  is the maximum size of all function definitions accessible within the expression.*

□

### Proof

In order to prove that every expression encountered by the higher order deforestation algorithm satisfies the given grammar, it is shown for each transformation rule that if the original expression to be transformed satisfies the given grammar, then any expressions which need to be further transformed will also satisfy this grammar.

□

**Lemma 5.9 (on size of expressions encountered during higher order deforestation)** *The size of all expressions described by the grammar  $dg^s(s, n)$  is bounded by  $s \times (n + 1)$ .*

□

## Proof

In order to prove that the size of all expressions described by the given grammar is bounded, it is shown that the size of each possible term in the grammar is bounded. The proof is by induction on the variables of the grammar.

□

## 6 Related Work

The first order deforestation algorithm described in [13] was extended to allow a restricted higher order facility by treating higher order functions as macros. These macros can have higher order arguments, but are not allowed to be recursive. This lack of recursion guarantees that all higher order macro calls can be expanded out to an equivalent first order expression at compile-time. Although these macros are not allowed to be recursive, they can still use a form of local recursion by introducing **where** expressions in which only first order functions are allowed to be recursive. For example, the following macro can be used to define the *map* function:

$$\begin{aligned} \mathit{map} &= \lambda f.\lambda xs.g\ xs \\ &\quad \mathbf{where} \\ g &= \lambda xs.\mathbf{case}\ xs\ \mathbf{of} \\ &\quad \quad \mathit{Nil} \quad \quad : \mathit{Nil} \\ &\quad \quad \mathit{Cons}\ x\ xs : \mathit{Cons}\ (f\ x)\ (g\ xs) \end{aligned}$$

This mechanism does not cover all uses of higher order functions. In particular, all data structures are first order, so it is not possible to define a list of functions using these macros. In the work described here, the deforestation algorithm has been reformulated so that it applies to all higher order functions directly. As a result, it is possible to transform expressions involving lists of functions. However, the non-linear function-type arguments in commonly used higher order functions such as *map*, *fold* and *filter* are linearised using these macros, so more useful transformations can be performed on expressions containing calls of these functions than when using the algorithm described in this paper.

An extension of deforestation to handle higher order functions is described in [2]. This extension involves transforming programs to remove higher order features. Not all higher order features can be removed by this transformation, but it produces a specialised form of higher order program. This specialised form of program can then be transformed to remove intermediate structures using an extension of the deforestation algorithm. Other higher order features may re-appear as a result of this deforestation, so they need to be removed again before the deforestation can continue. This problem could be avoided if the deforestation algorithm were reformulated to apply to all higher order functions directly, as is done in the work described in this paper. No separate transformation would then be required to remove the higher order features from a program. However, the algorithm used in [2] is applicable to a wider range of expressions than the algorithm described in this paper as some functions with non-linear arguments can be successfully transformed.

A reformulation of the deforestation algorithm which can be applied to higher order functional programs is described in [8]. The transformation rules are given as three mutually recursive functions, and are therefore not as intuitive as the rules given in this paper. More importantly, no higher order treeless form is defined for the algorithm in [8], so there is no guarantee that the algorithm will terminate. Also, there is no guarantee that the expression resulting from the transformation will be more efficient than the original expression.

Another reformulation of the deforestation algorithm which can be applied to all higher functional programs is described in [10]. The transformation rules are presented in a more straightforward manner than in [8] by identifying the next redex within the expression being transformed. However, no treeless form is defined in this paper either, so there is no guarantee that the algorithm will terminate or that the result will be more efficient. A similar reformulation is also given in [11] which does include proofs that the algorithm will terminate and that its result will be more efficient, but again no treeless form is defined.

A different technique for achieving much the same effect as deforestation is described in [4]. This technique involves defining functions which consume lists in terms of the function *foldr*, and functions which produce lists in terms of the function *build*. When calls of the functions *foldr* and *build* appear next to each other, they can be coalesced to avoid the production of an intermediate list. This technique applies only to functions which can be written efficiently in terms of *foldr* and *build*, and requires that calls of these functions appear next to each other. It is therefore applicable to a different set of expressions than the algorithm described in this paper.

## 7 Conclusion

In this paper, it has been shown how the deforestation algorithm given in [13] can be extended to handle higher order functional programs. It is argued that this extended algorithm is suitable for inclusion in an optimising compiler. Higher order treeless form is an easily recognised form of expression, and any function definition can easily be generalised so that it is in this form.

A prototype implementation of the higher order deforestation algorithm has been completed, and was used to transform the example programs in this paper. The inclusion of such an implementation in a functional language compiler would allow a more thorough evaluation of the worth of higher order deforestation.

Higher order treeless form could be extended to allow more useful transformations to be performed on commonly used higher order functions such as *map*, *fold* and *filter*, which are non-linear in their function-type arguments. These functions are a problem because the deforestation algorithm uses a call-by-name evaluation strategy, but the language on which it operates uses a call-by-need evaluation strategy. If the deforestation algorithm could be re-formulated in terms of a call-by-need evaluation strategy then this problem could be overcome. Research is continuing in this area.

## Bibliography

1. L. Augustsson. Compiling pattern matching. *Lecture Notes in Computer Science*, 201:368–381, 1985.
2. Wei-Ngan Chin. *Automatic Methods for Program Transformation*. PhD thesis, Imperial College, University of London, July 1990.
3. L. Damas and R. Milner. Principal type schemes for functional programs. In *Proceedings of the Ninth ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.
4. A. Gill, J. Launchbury, and S.L. Peyton Jones. A short cut to deforestation. In *Proceedings of the Sixth International Conference on Functional Programming Languages and Computer Architecture*, 1993.
5. R.J.M. Hughes. Supercombinators: A new implementation method for applicative languages. In *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 1–10, 1982.
6. R.J.M. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, April 1989.
7. T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Proceedings of the Workshop on Implementation of Functional Languages*, pages 165–180, February 1985.
8. S. Marlow and P. Wadler. Deforestation for higher-order functions. In *Proceedings of the Fifth Annual Glasgow Workshop on Functional Programming*, pages 154–165, July 1992.
9. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17:348–375, 1978.
10. K. Nielsen and M. H. Sorensen. Deforestation, partial evaluation, and evaluation orders. Submitted to *Principles of Programming Languages*, 1995.

11. D. Sands. Total correctness by local improvement in program transformation. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, January 1995.
12. P. Wadler. Efficient compilation of pattern matching. In S.L. Peyton Jones, editor, *The Implementation of Functional Programming Languages*, pages 78–103. Prentice Hall, 1987.
13. P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.