Chapter 1

Extending Higher-Order Deforestation: Transforming Programs to Eliminate Even More Trees

G.W. Hamilton

Abstract: In previous work, we have shown how Wadler’s original deforestation algorithm can be extended to handle higher-order programs. A higher-order treeless form of expression was defined to ensure the termination of this algorithm. Our higher-order algorithm was further extended by Seidl and Sørensen, and this extension was shown to remove some intermediate structures not removed by our algorithm (although our algorithm can also remove some intermediate structures not removed by their technique). In this paper, we show how our original algorithm can be further extended to remove the intermediate structures in the examples given by Seidl and Sørensen. We argue that, because our extended algorithm uses an easy to recognise treeless form, there is more transparency for the programmer in terms of the improvements which will be made. Also, unlike the algorithm of Seidl and Sørensen, our extended algorithm is guaranteed to result in no loss of efficiency. We argue that this is essential for any optimisation.

1.1 INTRODUCTION

The use of intermediate structures, lazy evaluation and higher-order functions in functional programming facilitates a more elegant and readable style of programming, but it often results in inefficient programs. One solution to this problem is to transform these programs into more efficient equivalent programs. A transformation algorithm called deforestation was proposed by Wadler [Wad90], which

1School of Computer Applications, Dublin City University, Dublin 9, Ireland
can eliminate intermediate structures from first-order functional programs. This algorithm defines a treeless form of definition in which no intermediate data structures are constructed by requiring that all function arguments and case selectors are variables. Every expression in which all functions have treeless definitions can be transformed by the deforestation algorithm to an equivalent treeless expression.

The first definition of a treeless form for deforestation of higher-order programs, and the associated proof of termination was given by Hamilton [Ham96], and a later similar formulation was given by Marlow [Mar96]. The treeless form defined is similar in spirit to the original first-order treeless form defined by Wadler. It is ensured that no intermediate data structures are created by requiring that all function arguments and case selectors are variables.

Seidl and Sørensen [SS97, SS98] give a further extension of our original higher-order deforestation algorithm which uses a constraint-based control-flow analysis to identify those terms which prevent deforestation from terminating, and these terms are extracted prior to transformation. This formulation will remove intermediate structures not removed by our original algorithm.

Example 1.1. Consider the program shown in Fig. 1.1.

\[
\text{foldr}\ c\ w\ (\text{foldr}\ c\ v\ u)
\]

\[
\text{where} \\
\quad c = \lambda x,\lambda xs,\text{Cons}\ x\ xs
\]

\[
\text{foldr} = \lambda f,\lambda a,\lambda xs,\text{case}\ xs\ of
\quad \text{Nil} & : a \\
\quad \text{Cons}\ x\ xs & : f\ x\ (\text{foldr}\ f\ a\ xs)
\]

\text{FIGURE 1.1. Example program}

This program appends the three lists \(u\), \(v\) and \(w\). \(v\) is appended to \(u\) to create an intermediate list, and then \(w\) is appended to this list. This program will be transformed into the one shown in Fig. 1.2 by the algorithm described in [SS97, SS98].

Our original higher-order deforestation algorithm will extract the term \(\text{foldr}\ f\ a\ xs\), and thus prevent this transformation. Thus the algorithm described in [SS97, SS98] will remove intermediate structures not removed by our original algorithm. However, the technique used in [SS97, SS98] also requires that constructors cannot have function-type arguments, which is not a requirement of our original algorithm. It will therefore not remove some structures which will be removed by our original algorithm (for an example, see [Ham01b]).

In this paper, we give an extension of our original higher-order deforestation algorithm (similar to the extension to the first-order algorithm given in [Ham93])
which can remove the intermediate structures in the examples given in [SS97, SS98], such as the one shown in Fig. 1.1. The extension involves identifying intermediate structures and accumulating parameters, and extracting these prior to transformation. We prove that this extended algorithm terminates and we show that, unlike the algorithm described in [SS97, SS98], it is guaranteed to result in no loss of efficiency.

The remainder of this paper is structured as follows. In Section 2, we define the higher-order language on which the described transformations are performed. In Section 3, we define extended higher-order treeless form, and in Section 4, the extended higher-order deforestation algorithm is defined. In Section 5, we state and prove the extended higher-order deforestation theorem, and Section 6 concludes.

1.2 LANGUAGE

In this section, we describe the language which will be used throughout this paper.

Definition 1.2 (Higher-Order Language). The language for which the described transformations are to be performed is a simple higher-order functional language as shown in Fig. 1.3. □

Programs in the language consist of an expression to evaluate and a set of function definitions. It is assumed that the language is typed using the Hindley-Milner polymorphic typing system [Mil78, DM82]. It is also assumed that the language is implemented using the full laziness technique described in [Hug82]. 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 [Joh85].

For the purposes of this paper, constants (e.g. numbers) and basic functions (+, *, >, square, 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
**FIGURE 1.3. Language grammar**

application must be saturated. Within case expressions of the form \( \text{case } e_0 \text{ of } p_1 : e_1 \mid \cdots \mid p_k : e_k \), \( e_0 \) is called the selector, and \( e_1 \ldots e_k \) are called the branches. 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 [Aug85, Wad87].

### 1.3 EXTENDED HIGHER-ORDER TREELESS FORM

Treeless form is a form of definition in which no intermediate structures are constructed. The reason why this form is important to the deforestation algorithm is that termination of the algorithm is ensured if all function definitions are in treeless form. In this section, an extended higher-order treeless form of expression is defined which extends the higher-order treeless form defined in [Ham96]. In the higher-order treeless form defined in [Ham96], expressions which are considered to be intermediate are blazed \( \Box \). These expressions are extracted prior to transformation and are transformed separately. The expressions which are considered to be intermediate are function arguments or case selectors which are not variables. However, some function arguments may appear directly in the result of the function, and are therefore not intermediate. We therefore extend the definition of intermediate structures as follows.

**Definition 1.3 (Intermediate Structure).** A structure is intermediate within an expression if it is used as the selector in a case expression during the evaluation of the expression.

Determining whether a structure is intermediate is not done statically; this is determined dynamically during transformation. Expressions which are blazed \( \Box \) are therefore extracted and transformed separately only if they ever appear within the selector of a case expression during transformation.
Example 1.4. The program given in Fig. 1.1 would be blazed as shown in Fig. 1.4.

\[
\begin{align*}
\text{foldr } c \ w (\text{foldr } c \ v \ u) \\
\text{where} \\
c & = \lambda x, x, \text{Cons} \ x \ x \\
f \text{oldr} & = \lambda f, x, a, x, \text{case} \ x \ x \ \text{of} \\
\text{Nil} & : a \\
\text{Cons} \ x \ x & : f \ x \ (\text{foldr} \ f \ a \ x) \ \Box
\end{align*}
\]

FIGURE 1.4. Example program with potentially intermediate structures blazed

The expression which is blazed \( \Box \) never actually appears as the selector in a case statement during transformation, and the program given in Fig. 1.2 is produced as a result of transformation.

This extension to the higher-order deforestation algorithm will not terminate for some expressions which contain no intermediate structures according to the above definition.

Example 1.5. Consider the program given in Fig. 1.5.

\[
\begin{align*}
\text{foldl } c \ \text{Nil} \ x \ s \\
\text{where} \\
c & = \lambda x, x, \text{Cons} \ x \ x \\
f \text{oldl} & = \lambda f, a, x, \text{case} \ x \ x \ \text{of} \\
\text{Nil} & : a \\
\text{Cons} \ x \ x & : \text{foldl} \ f \ (f \ x \ a) \ x
\end{align*}
\]

FIGURE 1.5. Example program with accumulating parameter

This program reverses the list \( x \). The size of the second parameter in the recursive call of foldl continually increases during the transformation, so the transformation fails to terminate. This situation occurs when a recursive function accumulates information in its parameters. We therefore define recursive function calls as follows.
Definition 1.6 (Recursive Function Call). A function call is recursive if it occurs within the definition of a function which the recursive function calls (either directly or indirectly).

Note that if there are $n$ functions, then at most $n$ functions must be inspected to determine whether each function is recursive. To check all functions is therefore $O(n^2)$. Accumulating parameters can now be defined as follows.

Definition 1.7 (Accumulating Parameter). An argument in a recursive function call is an accumulating parameter if it is not a variable.

Accumulating parameters are blazed ⊕ and are always extracted prior to transformation.

Example 1.8. The program given in Fig. 1.5 would be blazed as shown in Fig. 1.6.

```
foldl c Nil xs
where
  c  =  λx,λxs,Cons x xs
  foldl =  λf,λa,λxs, case xs of
           Nil  :  a
           Cons x xs :  foldl f (f x a) ⊕ xs
```

FIGURE 1.6. Example program with accumulating parameter blazed

Transformation of the program will now terminate, as the expression blazed ⊕ will be extracted prior to transformation.

It is essential for any optimisation that it does not result in any loss of efficiency. We therefore define the linearity of variables as follows.

Definition 1.9 (Linear). 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.

As in the higher-order treeless form defined in [Ham96], all non-linear variables are blazed ⊗ at their binding occurrence. Expressions cannot be substituted for variables which are blazed ⊗, and will be transformed separately. This ensures that expressions which are expensive to compute will not be duplicated. For example, the function foldr given in Fig. 1.1 is linear with respect to the variable $a$, but it is not linear with respect to the variable $f$. 

6
Example 1.10. The program in Fig. 1.1 would be blazed as shown in Fig. 1.7.  
With this restriction on non-linear variables, this program will be transformed to give the program shown in Fig. 1.8. We can see that, even though we cannot substitute the function \( c \) into the body of the function \( \text{foldr} \) as the variable \( f \) is non-linear, we have still removed all intermediate structures from this program. Extended higher-order treeless form can now be defined as follows.

**Definition 1.11 (Extended Higher-Order Treeless Form).** An expression is in extended higher-order treeless form if the following conditions are all satisfied:

1. all accumulating parameters are blazed \( \oplus \)
2. all other function arguments and \text{case} selectors are blazed \( \ominus \)
3. all non-linear variables are blazed \( \ominus \) at their binding occurrence
4. all function-type expressions are in extended higher-order treeless form
Note that this blazing need only be done once prior to transformation; the blazing of expressions does not change during transformation. The final restriction above was not a restriction in our original higher-order treeless form, but it is necessary to ensure that there is no loss of efficiency resulting from transformation (see [Ham01b]).

1.4 THE EXTENDED HIGHER-ORDER DEFORESTATION ALGORITHM

In this section, we present the higher-order deforestation algorithm. This is a set of transformation rules which attempt to convert a given expression into a higher order treeless equivalent. As in [SS97, SS98], we define these rules by identifying the next reducible expression (redex) within a given context. An expression which cannot be broken down into a redex and a context is called an observable. These are defined as follows.

Definition 1.12 (Redexes, Contexts and Observables). Redexes, contexts and observables are defined by the grammar shown in Fig. 1.9, where \( \text{red} \) ranges over redexes, \( \text{con} \) ranges over contexts and \( \text{obs} \) ranges over observables.

\[
\begin{align*}
\text{red} & ::= f \\
& \quad e \in \mathcal{E} \\
& \quad (\lambda v. e_0) \; e_1 \\
& \quad \text{case} \ (c \; e_0, \ldots, e_n) \; \text{of} \; p_1 : e_1^1, \ldots, p_k : e_k^k \\
& \quad \text{case} \ (v \; e_0, \ldots, e_n) \; \text{of} \; p_1 : e_1^1, \ldots, p_k : e_k^k
\end{align*}
\]

\[
\begin{align*}
\text{con} & ::= \hat{\text{e}} \\
& \quad \text{con} \; e \\
& \quad \text{case} \; \text{con} \; \text{of} \; p_1 : e_1, \ldots, p_k : e_k
\end{align*}
\]

\[
\begin{align*}
\text{obs} & ::= c \; e_1, \ldots, e_n \\
& \quad v \; e_1, \ldots, e_n \\
& \quad \lambda v. e
\end{align*}
\]

FIGURE 1.9. Grammar of redexes, contexts and observables

Every expression \( e \) is an observable or decomposes uniquely into a context \( c \) and redex \( r \) with \( e = c(r) \).

Definition 1.13 (Extended Higher-Order Deforestation Algorithm). The transformation rules for the extended higher-order deforestation algorithm are shown in Fig. 1.10.
These rules cover all possible kinds of expression (variable, constructor, lambda abstraction, function, application and case expression). In rules (1) - (3), the transformation rules are applied recursively to the sub-expressions of the expression being transformed. In rule (4), the function variable in an application is replaced with its body. In rule (5), an accumulating parameter is extracted from a lambda application and transformed separately. In rule (6), an argument in a lambda application is extracted and transformed separately if it is non-linear. In rule (7), an argument in a lambda application is substituted for the bound variable within the lambda body. In rule (8), an intermediate structure (an expression blazed \(\oplus\) which appears as the selector in a case expression) is extracted and transformed separately. In rule (9), the transformation rules are applied recursively to the sub-expressions of the expression being transformed. In rule (10), when the selector is a constructor application, pattern matching is applied and the case expression is replaced by the appropriate branch. This is where intermediate structures are actually removed by deforestation by avoiding the need for the intermediate constructor application. Rules (7) - (9) 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 [Wad90], the extended 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 call (this actually relies on the linearity constraint). A new function definition is therefore introduced before the application of rule (4). The right hand side of the function definition is the expression which was 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. This folding is defined more formally as follows.

**Definition 1.14 (Folding).** Transformation rule (4) must be changed as shown in Fig. 1.11 to define folding explicitly.

\[
T[e(f)] \phi = \begin{cases} 
  f' v_1 \ldots v_n, & \text{if } (f' = \lambda v'_1 \ldots v'_n . e') \in \phi \\
  \text{and } e(f) = e'[v_1/v'_1, \ldots, v_n/v'_n] \\
  f' v_1 \ldots v_n, & \text{otherwise}
\end{cases}
\]

where
\[
f \text{ is defined by } f = e' \\
f' = \lambda v'_1 \ldots v_n . (T[e(f)] \phi) \\
\phi' = \phi \cup \{f' = \lambda v'_1 \ldots v_n . e(f)\} \\
v_1, \ldots, v_n \text{ are the free variables in } e(f)
\]

**FIGURE 1.11.** Folding rule for extended higher-order deforestation

This rule is similar to the rules for first order deforestation given in [Chi90]. The additional parameter \( \phi \) contains the set of expressions which have been encountered before during the transformation and the associated function name which was introduced. This additional parameter must also be passed to all other transformation rules.

1.5 THE EXTENDED HIGHER-ORDER DEFORESTATION THEOREM

The extended higher-order deforestation theorem can now be stated as follows.

**Theorem 1.15 (Extended Higher-Order Deforestation Theorem).** Every program in which each non-linear variable is blazed \( \Box \) at its binding occurrence, and in which all function-type expressions are in extended higher-order treeless form can be transformed to an equivalent extended higher-order treeless program without loss of efficiency. \( \Box \)
We concentrate here on proving the termination of the algorithm. A proof of the correctness of the algorithm can be found in [San95], and a proof that the algorithm results in no loss of efficiency can be found in [Ham01b] (this relies on the fact that the language is implemented using the full laziness technique described in [Hug82]). In order to prove that the algorithm always terminates, we show that there is a bound on the size of expressions which are encountered during transformation. We define the size of an expression to be the sum of the size of all the sub-expressions it contains plus one. For example, \( S[e_1, \ldots, e_n] = 1 + S[e_1] + \ldots + S[e_n] \). Note that this definition incorporates both the depth and width of expressions, rather than dealing with the depth and width of expressions separately, as was previously done in [Ham96, Mar96].

**Lemma 1.16 (On Size of Expressions).** The size of all expressions encountered by the extended higher-order deforestation algorithm is bounded by \( \text{size}_{\text{init}} \times \text{num}_f \times \text{size}_f \times \text{args}_f \), where \( \text{size}_{\text{init}} \) is the size of the initial expression to be transformed, \( \text{num}_f \) is the number of function definitions in the program being transformed, \( \text{size}_f \) is the maximum size of any function definition, and \( \text{args}_f \) is the maximum number of arguments in any function definition.

**Proof.** We prove that expressions encountered by the algorithm must always be of the form \( e^{(f,s)} \), where \( e^{(f,s)} = e[v_1, \ldots, v_n] \), \( e^{(f,s)} = e^{(f,-1)} \), and \( e^{(0,s)} = v \), where \( f \leq \text{num}_f, s \leq \text{size}_{\text{init}}, n \leq \text{args}_f \) and \( S[e] \leq \text{size}_f \). Details of this proof can be found in [Ham01a].

### 1.6 Conclusion

In this paper, it has been shown how the higher-order deforestation algorithm given in [Ham96] can be extended to remove more intermediate structures. A previous extension of our higher-order algorithm was given by Seidl and Sørensen [SS97, SS98], which was shown to remove some intermediate structures not removed by our original algorithm. Our extended algorithm will remove the intermediate structures in the examples given by Seidl and Sørensen. Their algorithm will still remove some intermediate structures which are not removed by our own technique, although the converse is also true. For example, Seidl and Sørensen require that constructors cannot have function-type arguments, which is not a requirement of our extended algorithm. We argue that there is more transparency for the programmer in terms of the improvements which will be made using our extended algorithm, because it uses an easy to recognise treeless form. The generalisation technique used by Seidl and Sørensen is quite complicated and expensive (\( O(n^4) \) for a program of size \( n \)), compared to our own (which is \( O(n^2) \)). Also, unlike the algorithm of Seidl and Sørensen, our extended algorithm is guaranteed to result in no loss of efficiency. We argue that this is essential for any optimisation.
REFERENCES


