Fusion in Practice*

Diederik van Arkel, John van Groningen, and Sjaak Smetsers

Computing Science Institute
University of Nijmegen
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands
diederik@cs.kun.nl, johnvg@cs.kun.nl, sjakie@cs.kun.nl

Abstract. Deforestation was introduced to eliminate intermediate data structures used to connect separate parts of a functional program together. Fusion is a more sophisticated technique, based on a producer-consumer model, to eliminate intermediate data structures. It achieves better results. In this paper we extend this fusion algorithm by refining this model, and by adding new transformation rules. The extended fusion algorithm is able to deal with standard deforestation, but also with higher-order function removal and dictionary elimination. We have implemented this extended algorithm in the Clean 2.0 compiler.

1 Introduction

Static analysis techniques, such as typing and strictness analysis are crucial components of state-of-the-art implementations of lazy functional programming languages. These techniques are employed to determine properties of functions in a program. These properties can be used by the programmer and also by the compiler itself. The growing complexity of functional languages like Haskell [Has92] and Clean [Cle13,Cle20] requires increasingly sophisticated methods for translating programs written in these languages into efficient executables. Often these optimization methods are implemented in an ad hoc manner: new language features seem to require new optimization techniques which are implemented simultaneously, or added later when it is noticed that the use of these features leads to inefficient code. For instance, type classes require the elimination of dictionaries, monadic programs introduce a lot of higher-order functions that have to be removed, and the intermediate data structures that are built due to function composition should be avoided.

In Clean 1.3 most of these optimizations were implemented independently. They also occurred at different phases during the compilation process making it difficult to combine them into a single optimization phase. The removal of auxiliary data structures was not implemented at all. This meant that earlier optimisations did not benefit from the transformations performed by later optimisations.

This paper describes the combined method that has been implemented in Clean 2.0 to perform various optimizations. This method is based on Chin's

* This work was supported by STW as part of project NWI.4411

fusion algorithm [Chin94], which in its turn was inspired by Wadler's *deforestation algorithm* [Wad88,Fer88]. The two main differences between our method and Chin's fusion are (1) we use a more refined analysis to determine which functions can safely be fused, and (2) our algorithm has been implemented as part of the Clean 2.0 compiler which makes it possible to measure its effect on real programs (See Section 6).

The paper is organized as follows. We start with a few examples illustrating what types of optimizations can be performed (Section 2). In Section 3 we explain the underlying idea for deforestation. Section 4 introduces a formal language for denoting functional programs. In Section 5 we present our improved fusion algorithm and illustrate the effectiveness of this algorithm with a few example programs (Section 6). We conclude with a discussion of related work (Section 7) and future research (Section 8).

2 Overview

This section gives an overview of the optimizations that are performed by our improved fusion algorithm. Besides traditional deforestation, we illustrate so-called dictionary elimination and general higher-order function removal. We also indicate the 'pitfalls' that may lead to non-termination or to duplication of work, and present solutions to avoid these pitfalls.

The transformation rules of the fusion algorithm are defined by using a core language (Section 4). Although there are many syntactical differences between the core language and Clean we distinguish these languages more explicitly by using a sans serif style for core programs and a `typewriter` style for Clean programs.

2.1 Deforestation

Deforestation attempts to transform a functional program which uses intermediate data structures into one which does not. Note that these data structures can be of arbitrary type, they are not restricted to lists. These intermediate data structures are common in lazy functional programs as they enable modularity. For example, the function `any`, which tests whether any element of a list satisfies a given predicate, could be defined as:

```
any p xs = or (map p xs)
```

Here `map` applies `p` to all elements of the list `xs` yielding an intermediate list of boolean values. The function `or` combines these values to produce a single boolean result. This style of definition is enabled by lazy evaluation, the elements of the intermediate list of booleans are produced one at a time and immediately consumed by the `or` function, thus the function `any` can run in constant space. However this definition is still wasteful, each element of the intermediate list has to be allocated, filled, taken apart, and garbage collected.

If deforestation is successful it transforms the definition of `any` into the following efficient version:

```
any p [] = False
any p [x:xs] = p x || any p xs
```

The following example is given in both [Chin94] and [Wad88]. It elegantly shows that a programmer no longer needs to worry about annoying questions such as "Which of the following two expressions is more efficient?"

`append (append a b) c` or `append a (append b c)`

where `append` is defined as

```
append []     ys = ys
append [x:xs] ys = [x:append xs ys]
```

Experienced programmers almost automatically use the second expression, but from a more abstract point of view there seems to be no difference.

Deforestation as well as fusion will transform the left expression into the expression `app_app a b c` and introduce a new function in which the two applications of `append` are combined:

```
app_app []     b c = append b c
app_app [x:xs] b c = [x:app_app xs b c]
```

Transforming the right expression leads to essentially the same function as `app_app` both using Wadler's deforestation and by Chin's fusion. However, this saves only one evaluation step compared to the original expression at the cost of an extra function. Our fusion algorithm transforms the left expression just as deforestation or fusion would, but it leaves the right expression unchanged.

The major difficulty with this kind of transformation is to determine which applications can be fused (or deforested) safely. Without any precautions there are many situations in which the transformation will not terminate. Therefore it is necessary to formulate proper criteria that, on the one hand, guarantee termination, and on the other hand, do not reject too many fusion candidates. Besides termination, there is another problem that has to be dealt with: the transformation should not introduce repeated computations, by duplicating redexes. We will have a closer look at non-termination and redex duplication in Section 5.

2.2 Type classes and dictionary removal

Type classes or ad-hoc polymorphism are generally considered to be one of the most powerful concepts of functional languages [Wad89]. The advantages are illustrated in the following example in which an instance of equality for lists is declared. Here we use the Clean syntax (deviating slightly from the Haskell notation).

```
instance == [a] | == a
where
    (==) []     []     = True
    (==) [x:xs] [y:ys] = x == y && xs == ys
    (==) _      _      = False
```

With type classes one can use the same name (==) for defining equality over all kinds of different types. In the body of an instance one can use the overloaded operation itself to compare substructures, i.e. it is not necessary to indicate the difference between both occurrences of `==` in the right-hand side of the second alternative. The translation of such instances into 'real' functions is easy: Each *context restriction* (in our example `| == a`) is converted into a *dictionary* argument containing the concrete version of the overloaded class. For the equality example this leads to

```
eqList eq []     []      = True
eqList eq [x:xs] [y:ys]  = eq x y && eqList eq xs ys
eqList eq _      _       = False
```

An application of `==` to two lists of integers, e.g. `[1,2] == [1,2]`, is replaced by an expression containing the list version of equality parameterized with the integer dictionary of the equality class, `eqList eqInt [1,2][1,2]`.

Applying this simple compilation scheme introduces a lot of overhead which can be eliminated by specializing `eqList` for the `eqInt` dictionary as shown below

```
eqListeqInt []     []      = True
eqListeqInt [x:xs] [y:ys]  = eqInt x y && eqListeqInt xs ys
eqListeqInt _      _       = False
```

In Clean 1.3 the specialization of overloaded operations within a single module was performed immediately, i.e. dictionaries were not built at all, except for some rare, exotic cases. These exceptions are illustrated by the following type declaration (taken from [Oka98])

```
::  Seq a = Nil | Cons a (Seq [a])
```

Defining an instance of `==` for `Seq a` is easy, specializing such an instance for a concrete element type cannot be done. The compiler has to recognize such situations in order to avoid an infinite specialization loop.

In Clean 2.0 specialization is performed by the fusion algorithm. The handling of infinite specialization does not require special measures as the functions involved will be marked as unsafe by our fusion analysis. Moreover dictionaries do not contain unevaluated expressions (*closures*), so copying dictionaries can never duplicate computations. This means that certain requirements imposed by the fusion algorithm can be relaxed for dictionaries. In the remainder of the paper we leave out the treatment of dictionaries because besides this relaxation of requirements it very much resembles the way other constructs are analyzed and transformed.

2.3 Higher-order function removal

A straightforward treatment of higher-order functions introduces overhead both in time and space. E.g. measurements on large programs using a monadic style of programming show that such overhead can be large; see section 6.

In [Wad88] Wadler introduces *higher-order macros* to elimate some of this overhead but this method has one major limitation: these macros are not considered first class citizens. Chin has extended his fusion algorithm so that it is able to deal with higher-order functions. We adopt his solution with some minor refinements.

So called accumulating parameters are a source of non-termination. For example, consider the following function definitions:

```
twice f x = f (f x)

f g = f (twice g)
```

The parameter of `f` is accumulating: the argument `twice g` in the recursive call of `f` is 'larger' than the original argument. Trying to fuse `f` with `inc` (for some producer `inc`) in the application `f inc` will lead to a new application of the form `f twice_inc`. Fusing this one leads to the expression `f twice_twice_inc` and so on.

Partial function applications should also be treated with care. At first one might think that it is safe to fuse an application `f (g E)` in which the arity of `f` is greater than one and the subexpression `g E` is a redex. This fusion will combine `f` and `g` into a single function, say `f_g`, and replace the original expression by `f_g E`. This, however, is dangerous if the original expression was shared, as shown by the following function `h`:

```
h   = z 1 + z 2
    where z = f (g E)
```

This function is not equivalent to the version in which `f` and `g` have been fused:

```
h   = z 1 + z 2
    where z = f_g E
```

Here the computation encoded in the body of `g` will be performed twice, as compared to only once in the original version.

2.4 Optimizing generic functions

Generic programming allows the programmer to write a function once and use it for different types. It relieves the programmer from having to define new instances of common operations each time he declares a new data type. The idea is to consider types as being built up from a small fixed set of type constructors and to specify generic operations in terms of these constructors. In Clean 2.0, for example, one can specify all instances of equality by just a few lines of fairly obvious code:

```
generic eq a :: a a -> Bool

eq{|UNIT|}          x         y             = True
```

```
eq{|PAIR|}   eqx eqy (PAIR x y)(PAIR x' y') = eqx x x' && eqy y y'
eq{|EITHER|} eql eqr (LEFT l)  (LEFT l')    = eql l l'
eq{|EITHER|} eql eqr (RIGHT r) (RIGHT r')   = eqr r r'
eq{|EITHER|} eql eqr _ _                    = False
```

Here `UNIT`, `PAIR` and `EITHER` are the fixed generic types. With the aid of this generic specification, the compiler is able to generate instances for any algebraic data type. The idea is to convert an object of such a data type to its generic representation (this encoding follows directly from the definition of the data type), apply the generic operation to this converted object and, if necessary, convert the object back to a data type. For a comprehensive description of how generics can be implemented, see [Ali01] or [Hin00].

Without any optimizations one obtains operations which are very inefficient. The conversions and the fact that generic functions are higher-order functions (e.g. the instance of `eq` for `PAIR` requires two functions as arguments, `eqx` and `eqy`) introduce a lot of overhead. The combined data and higher-order fusion is sufficient to get rid of almost all intermediate data and higher-order calls, leading to specialized operations that are usually as efficient as the hand coded versions. To achieve this, only some minor extensions of the original fusion algorithm were needed.

3 Introduction to fusion

The underlying idea for transformation algorithms like Wadler's deforestation or Chin's fusion is to combine nested applications of functions of the form $F(\ldots, G\vec{E}, \ldots)$[1] into a single application $F_i G(\ldots, \vec{E}, \ldots)$. This is achieved by performing a sequence of unfold steps of both F and G. An unfold step is the substitution of a function body for an application of that function, wheareas a fold step performs the reverse process. Of course, if one of the functions involved is recursive this sequence is potentially infinite. To avoid this it is necessary that during the sequence of unfold steps an application is reached that has been encountered before. In that case one can perform the crucial fold step to achieve termination. But how do we know that we will certainly reach such an application?

Wadler's solution is to define the notion of *treeless form.* If the above F and G are treeless it is guaranteed that no infinite unfolding sequences will occur. However, Wadler does not distinguish between F and G. Chin recognizes that the roles of these functions in the fusion process are different. He comes up with the so called producer-consumer model: F plays the role of *consumer*, consuming data through one of its arguments, whereas G acts as a *producer*, producing data via its result. Separate safety criteria can then be applied for the different roles.

Although Chin's criterion indicates more fusion candidates than Wadler's, there are still cases in which it appears to be too restrictive. To illustrate these shortcomings we first repeat Chin's notion of safety: A function F is a safe

[1] We write $\vec{E}$ as shorthand for $(E_1, \ldots, E_n)$

consumer in its i^{th} argument if all recursive calls of F have either a variable or a constant on the i^{th} parameter position, otherwise it is *accumulating* in that argument. A function G is a safe producer if none of its recursive calls appears on a consuming position.

One of the drawbacks of the safety criterion for consumers is that it indicates too many consuming parameters, and consequently it limits the number of producers (since the producer property negatively depends on the consumer property). As an example, consider the following definition for `flatten`:

```
flatten [] = []
flatten [x:xs] = append x (flatten xs)
```

According to Chin, the `append` function is consuming in both of its arguments. Consequently, the `flatten` function is not a producer, for, its recursive call appears on a consuming position of `append`. Wadler will also reject `flatten` because its definition is not treeless.

In our definition of consumer we will introduce an auxiliary notion, called *active arguments*, that filters out the arguments that will not lead to a fold step, like the second argument of `append`. If `append` is no longer consuming in its second argument, `flatten` becomes a decent producer.

Chin also indicates superfluous consuming arguments when we are not dealing with a single recursive function but with a set of mutually recursive functions. To illustrate this, consider the unary functions `f` and `g` being mutually recursive as follows:

```
f x = g x
g x = f (h x)
```

Now `f` is accumulating and `g` is not (e.g. `g`'s body contains a call to `f` with an accumulating argument whereas `f`'s body just contains a simple call to `g`). Although `g` is a proper consumer by Chin's definition, it makes no sense to fuse an application of `g` with a producer, for this producer will be passed to `f` but cannot be fused with `f`. Again no fold step can take place. Unfortunately, by considering `g` as consuming, any function of which the recursive call appears as an argument of `g` will be rejected as a producer. There is no need for that, and therefore we indicate both `f` and `g` as non-consuming.

4 Syntax

We shall formulate the fusion algorithm with respect to a 'core language' which captures the essential aspects of lazy functional languages such as pattern matching, sharing and higher-order functions.

Functional expressions are built up from applications of function symbols F and data constructors C. Pattern matching is expressed by a construction case $\cdots$ of $\cdots$. In function definitions, one can express pattern matching with respect to one argument at a time. This means that compound patterns are decomposed into nested ('sequentialized') case expressions. Sharing of objects is expressed by a let construction and higher-order applications by an $\mathsf{@}$. We do not allow functions that contain case expressions as nested subexpressions on the right-hand side, i.e. case expressions can only occur at the outermost level. And as in [Chin94], we distinguish so called *g-type functions* (starting with a single pattern match) and *f-type functions* (with no pattern match at all).

Definition 1. (i) *The set of* expressions *is defined by the following grammar. Below, x ranges over variables, C over constructors and F over function symbols.*

$$\begin{aligned} E ::= {} & x \\ \mid {} & C(E_1,\ldots,E_k) \\ \mid {} & F(E_1,\ldots,E_k) \\ \mid {} & \mathsf{let}\ \vec{x} = \vec{E}\ \mathsf{in}\ E' \\ \mid {} & \mathsf{case}\ E\ \mathsf{of}\ P_1|E_1 \ldots P_n|E_n \\ \mid {} & E\ \mathsf{@}\ E' \\ P ::= {} & C(x_1,\ldots,x_k) \end{aligned}$$

(ii) *The set of* free variables *(in the obvious sense) of E is denoted by* $\mathrm{FV}(E)$*. An expression E is said to be* open *if* $\mathrm{FV}(E) \neq \emptyset$*, otherwise E is called* closed.

(iii) *A* function definition *is an equation of the form*

$$F(x_1,\ldots,x_k) = E$$

where all the x_i's are disjoint and $\mathrm{FV}(E) \subseteq \{x_1,\ldots,x_k\}$.

The semantics of the language is call-by-need.

5 Fusion Algorithm

5.1 Consumers

We start by defining the supporting notions of active and accumulating.

We say that an occurrence of variable x in E is *active* if x is either a pattern matching variable ($\mathsf{case}\ x\ \mathsf{of}\ \ldots$), a higher-order variable ($x\ \mathsf{@} \ldots$), or x is used as an argument on an active position of a function. The intuition here is to mark those function arguments where fusion can lead to a fold step or further transformations. This definition ensures that for example the second argument of append is not regarded as consuming.

We define the notions of 'active occurrence' $actocc(x, E)$ and 'active position' $act(F)_i$ simultaneously as the least solution of some predicate equations.

Definition 2. (i) *The predicates actocc and act are specified by mutual induction as follows.*

$$\begin{array}{ll}
actocc(x, y) & = true,\ if\ y = x \\
 & = false,\ otherwise \\
actocc(x, F\vec{E}) & = true,\ if\ for\ some\ i\text{: } E_i = x \wedge act(F)_i \\
 & = \bigvee_i actocc(x, E_i),\ otherwise \\
actocc(x, C\vec{E}) & = \bigvee_i actocc(x, E_i) \\
actocc(x, \mathsf{case}\ E\ \mathsf{of}\ \ldots P_i | E_i \ldots) & = true,\ if\ E = x \\
 & = \bigvee_i actocc(x, E_i),\ otherwise \\
actocc(x, \mathsf{let}\ \vec{x} = \vec{E}\ \mathsf{in}\ E') & = actocc(x, E') \vee \bigvee_i actocc(x, E_i) \\
actocc(x, E\ \texttt{@}\ E') & = true,\ if\ E = x \\
 & = actocc(x, E) \vee actocc(x, E'),\ otherwise
\end{array}$$

Moreover, for each function F, defined by $F\vec{x} = E$

$$act(F)_i \Leftrightarrow actocc(x_i, E)$$

(ii) *We say that F is* active *in argument i if $act(F)_i$ is true.*

The notion of *accumulating parameter* as introduced by [Chin94] is used to detect potential non-termination of fusion as we could see in the example in section 2.3. Our definition is a slightly modified version to deal with mutually recursive functions as indicated in 3.

Definition 3. *Let $F_1, \ldots, F_n$ be a set of mutually recursive functions with respective right-hand sides $E_1, \ldots, E_n$. The function F_j is* accumulating *in its i^{th} parameter if either*
(1) there exists a right-hand side E_k containing an application $F_j(\ldots, E_i', \ldots)$ in which E_i' is open and not just an argument or a pattern variable, or
(2) the right-hand side of F_j, E_j, contains an application $F_k(\ldots, E_l', \ldots)$ such that $E_l' = x_i$ and F_k is accumulating in l.

The first requirement corresponds to Chin's notion of accumulating parameter. The second requirement will prevent functions that recursively depend on other accumulating functions from being regarded as non-accumulating.

Combining the notions of active and accumulating leads to the notion of consuming, indicating that fusion is both interesting and safe for that parameter.

Definition 4. *A function F is* consuming *in its i^{th} parameter if it is both non-accumulating and active in i.*

5.2 Producers

The notion of *producer*, indicating that fusion will terminate for such a function as producer, is also taken from [Chin94] here with a minor adjustment to deal with constructors. First we define *producer* for functions

Definition 5. *Let $F_1, \ldots, F_n$ be a set of mutually recursive functions. These functions are called* producers *if none of their recursive calls (in the right-hand sides of their definitions) occurs on a consuming position.*

Now we extend the definition to application expressions

Definition 6. *An application of S e.g. $S(E_1, \ldots, E_k)$ is a producer if:*

1. $\mathsf{arity}(S) > k$, *or*
2. *S is a constructor*
3. *S is a producer function*

5.3 Linearity

The final notion we require is that of *linearity* which is used to detect potential duplication of work. It is unchanged from the original definition as introduced by Chin.

Definition 7. *Let F be a function with definition $F(x_1, \ldots, x_n) = E$. The function F is* linear *in its i^{th} parameter if*
(1) F is an f-type function and x_i occurs at most once in E, or
(2) F is a g-type function and x_i occurs at most once in each of the branches of the top-level case.

5.4 Transformation rules

Our adjusted version of the fusion algorithm consists of one general transformation rule, dealing with all expressions, and three auxiliary rules for function applications, higher-order application, and for case expressions. The idea is that during fusion both consumer and producer have to be unfolded and combined. This combination forms the body of the newly generated function. Sometimes however, it appears to be more convenient if the unfold step of the consumer could be undone, in particular if the consumer and the producer are both g-type functions. For this reason we supply some of the case expressions with the function symbol to which it corresponds (case_F). Note that this correspondence is always unique because g-type functions contain exactly one case on their right-hand side.

We use F_iS as a name for the function that results from fusing F that is consuming in its i_{th} argument, with producer S. Suppose F is defined as $F\vec{x} = E$. Then the resulting function is defined, distinguishing two cases

1. S is a fully applied function and F is a g-type function: the resulting function consists of the unfoldings of F and S. The top-level case is annotated as having come from F.
2. Otherwise the resulting function consists of the unfolding of F with the application of S substituted for formal argument i.

Note that each F_iS is generated only once.

Definition 8. *Rules for introducing fused functions.*

$$
\begin{array}{ll}
F_iG(x_1,\dots,x_{i-1},y_1,\dots,y_m,x_{i+1},\dots,x_n) & \\
\quad = \mathcal{T}[\![E[E'/x_i]]\!], & \textit{if } \mathsf{arity}(G) = m \textit{ and } F \textit{ is a g-type function} \\
 & \textit{where } G\vec{y} = E' \\
\quad = \mathcal{T}[\![E[G(y_1,\dots,y_m)/x_i]]\!], & \textit{otherwise} \\
F_iC(x_1,\dots,x_{i-1},y_1,\dots,y_m,x_{i+1},\dots,x_n) & \\
\quad = \mathcal{T}[\![E[C(y_1,\dots,y_m)/x_i]]\!] &
\end{array}
$$

We use F^+ as a name for the function that results from raising the arity of F by one. The $\mathcal{R}$-rules raise the arity of an expression by propagating the application of the extra argument y through the expression E.

Definition 9. *Rules for arity raising*

$$
\begin{array}{l}
F^+(x_1,\dots,x_n,y) \\
\;= \mathcal{T}[\![\mathcal{R}_y[\![E]\!]]\!] \\
\\
\mathcal{R}_y[\![\mathsf{let}\ \vec{x} = \vec{E}\ \mathsf{in}\ E']\!] \\
\;= \mathsf{let}\ \vec{x} = \vec{E}\ \mathsf{in}\ \mathcal{R}_y[\![E']\!] \\
\mathcal{R}_y[\![\mathsf{case}\ E\ \mathsf{of}\ \dots P_i|E_i\dots]\!] \\
\;= \mathsf{case}\ E\ \mathsf{of}\ \dots P_i|\mathcal{R}_y[\![E_i]\!]\dots \\
\mathcal{R}_y[\![E]\!] \\
\;= E\ @\ y
\end{array}
$$

The $\mathcal{T}$-rules recursively apply the transformation to all parts of the expression and invokes the appropriate auxiliary rules. These are the $\mathcal{F}$-rule for function applications which applies the safety criteria for fusion. The $\mathcal{H}$-rules for higher-order applications which replace higher-order applications by ordinary applications, using the arity-raised version of the applied function when necessary. And finally, the $\mathcal{C}$-rules for case expressions. The first alternative applies the fold-step where possible, the second alternative eliminates cases where the branch to be taken is known. The third alternative resolves nested cases by undoing the unfold step of F. A minor improvement can be obtained by examining the expression E'_i, if this expression starts with a constructor it is better to perform the pattern match instead.

Definition 10. *Transformation rules for first and higher-order expressions*

$$
\begin{array}{ll}
\mathcal{T}[\![x]\!] & = x \\
\mathcal{T}[\![C\vec{E}]\!] & = C\mathcal{T}[\![\vec{E}]\!] \\
\mathcal{T}[\![F\vec{E}]\!] & = \mathcal{F}[\![F\mathcal{T}[\![\vec{E}]\!]]\!] \\
\mathcal{T}[\![\mathsf{case}\ E\ \mathsf{of}\ \dots P_i|E_i\dots]\!] & = \mathcal{C}[\![\mathsf{case}\ \mathcal{T}[\![E]\!]\ \mathsf{of}\ \dots P_i|E_i\dots]\!] \\
\mathcal{T}[\![\mathsf{let}\ \vec{x} = \vec{E}\ \mathsf{in}\ E']\!] & = \mathsf{let}\ \vec{x} = \mathcal{T}[\![\vec{E}]\!]\ \mathsf{in}\ \mathcal{T}[\![E']\!] \\
\mathcal{T}[\![E\ @\ E']\!] & = \mathcal{H}[\![\mathcal{T}[\![E]\!]\ @\ \mathcal{T}[\![E']\!]]\!] \\
\mathcal{T}[\![\vec{E}]\!] & = (\mathcal{T}[\![E_1]\!],\dots,\mathcal{T}[\![E_n]\!])
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{F}[\![F(E_1,\ldots,E_i,\ldots,E_m)]\!] & \\
 & = \mathcal{F}[\![F_i S(E_1,\ldots,E_{i-1},E'_1,\ldots,E'_n,E_{i+1}\ldots,E_m)]\!], \\
 & \quad \textit{if: for some } i \textit{ with } E_i = S(E'_1,\ldots,E'_n) \\
 & \quad \textit{1) } F \textit{ is consuming in } i \\
 & \quad \textit{2) } S(E'_1,\ldots,E'_n) \textit{ is a producer} \\
 & \quad \textit{3) } \mathsf{arity}(S) \neq n \\
 & \quad \textit{or} \\
 & \quad \textit{1) } F \textit{ is both consuming and linear in } i \\
 & \quad \textit{2) } \mathsf{arity}(F) = m \textit{ and } \mathsf{arity}(S) = n \\
 & \quad \textit{3) } S(E'_1,\ldots,E'_n) \textit{ is a producer} \\
 & \quad\quad \textit{or has a higher order type} \\
 & = F(E_1,\ldots,E_i,\ldots,E_m), \textit{ otherwise} \\
 & \\
\mathcal{H}[\![C(E_1,\ldots,E_k) \mathbin{@} E]\!] & = C(E_1,\ldots,E_k,E) \\
\mathcal{H}[\![F(E_1,\ldots,E_k) \mathbin{@} E]\!] & = \mathcal{F}[\![F(E_1,\ldots,E_k,E)]\!], \textit{ if } \mathsf{arity}(F) > k \\
 & = \mathcal{F}[\![F^+(E_1,\ldots,E_k,E)]\!], \textit{ otherwise} \\
 & \\
\mathcal{C}[\![\mathsf{case}_F\, G(E_1,\ldots,E_n) \text{ of } \ldots]\!] & \\
 & = \mathcal{F}[\![F(x_1,\ldots,x_{i-1},G(E_1,\ldots,E_n),x_{i+1},\ldots,x_n)]\!] \\
 & \quad \textit{where} \\
 & \quad\quad\quad F(x_1,\ldots,x_n) = \mathsf{case}\ x_i \text{ of } \ldots \\
\mathcal{C}[\![\mathsf{case}_F\, C_i(E_1,\ldots,E_n) \text{ of } \ldots C_i(x_1,\ldots,x_n)|E'_i\ldots]\!] & \\
 & = \mathcal{T}[\![E'_i[E_1/x_1,\ldots,E_n/x_n]]\!] \\
\mathcal{C}[\![\mathsf{case}_F(\mathsf{case}\ E \text{ of } \ldots P_i|E'_i\ldots) \text{ of } \ldots]\!] & \\
 & = \mathsf{case}\ \mathcal{T}[\![E]\!] \text{ of } \ldots P_i|E''_i\ldots \\
 & \quad \textit{where } E''_i = \mathcal{F}[\![F(x_1,\ldots,x_{i-1},\mathcal{T}[\![E'_i]\!],x_{i+1},\ldots,x_n)]\!] \\
 & \quad \textit{and} \\
 & \quad\quad\quad F(x_1,\ldots,x_n) = \mathsf{case}\ x_i \text{ of } \ldots \\
\mathcal{C}[\![\mathsf{case}_F\, x \text{ of } \ldots P_i|E_i\ldots]\!] & = \mathsf{case}_F\, x \text{ of } \ldots P_i|\mathcal{T}[\![E_i]\!]\ldots
\end{array}
$$

6 Examples

We now present a few examples of fusion using the adjusted transformation rules.

6.1 Deforestation

We start with a rather trivial example involving the functions Append, Flatten and Reverse. The first two functions have been defined earlier. The definition of Reverse uses a helper function with an explicit accumulator:

$$
\begin{array}{lll}
\mathsf{Reverse}(l) & = \mathsf{Rev}(l,\mathsf{Nil}) & \\
 & & \\
\mathsf{Rev}(l,a) & = \mathsf{case}\ l\ \mathsf{of} & \\
 & \quad \mathsf{Nil} & |\ a \\
 & \quad \mathsf{Cons}(x,xs) & |\ \mathsf{Rev}(xs,\mathsf{Cons}(x,a)) \\
 & & \\
\mathsf{test}(l) & = \mathsf{Reverse}(\mathsf{Flatten}(l)) &
\end{array}
$$

The result of applying the transformation rules to the function test is shown below.

```
test(l)            = ReverseFlatten(l)

ReverseFlatten(l) = RevFlatten(l, Nil)

RevFlatten(l, r)   = case l of
                       Nil          | r
                       Cons(x, xs) | RevAppendFlatten(x, xs, r)

RevAppendFlatten(xs, l, r)
                   = case xs of
                       Nil          | RevFlatten(l, r)
                       Cons(x, xs) | RevAppendFlatten(xs, l, Cons(x, r))
```

One can give an alternative definition of Reverse using the standard higher-order *foldl* function. Transforming test then results in two mutually recursive functions that are identical to the auxiliary functions generated for the original definition of Reverse except for the order of the parameters.

Fusion appears to be much less successful if the following direct definition of reverse is used:

```
Reverseacc(l) = case l of
                  Nil          | Nil
                  Cons(x, xs) | Append(Reverseacc(xs), Cons(x, Nil))
```

Now Reverseacc is combined with Flatten but the fact that Reverseacc itself is not a producer (the recursive occurrence of this function appears on a consuming position) prevents the combination of Reverseacc and Flatten from being deforested completely. In general combinations of standard list functions (except for *accumulating* functions, such as Reverse), e.g. Sum(Map Inc(Take(n, Repeat(1)))) are transformed into a single function that generates no list at all and that does not contain any higher-order function applications.

6.2 Results

As a practical test of the effectiveness of the adjusted fusion algorithm we applied the fusion algorithm to several programs. The following test programs were used: `jpeg`[Fok95], `pseudoknot`[Har94] and the programs from [Har93], except `listcopy`. These programs were all ported to Clean 2.0. We computed the speedup by dividing the execution time without fusion by the execution time with fusion. Because the compiler does not support cross module optimisation, we copied frequently used standard list functions from the standard library to the module(s) of the program. The speedups for these programs are also shown in the table, but only if the speedup is not the same. In all cases specialization of overloaded functions was enabled. To show the effect of the specialization of overloaded functions we have run two small test programs: `mergesort` and `nfib`.

The optimisation of generic functions was tested with a small program converting arbitrary objects to and from the generic representation that has been used in [Ach02]. The largest test program is the Clean compiler itself. For the compiler the improvements were almost entirely caused by better optimization of a few modules that use a monadic style, and not by removal of intermediate data structures using deforestation. For this reason we have included the effect of fusion on these 'monadic' modules as a separate result. The results are summarized in the following table.

program	*speedup*	*added functions*	*speedup*
`jpeg`	1.34	`map`,`sum`	1.77
`pseudoknot`	1.11	`++`, `map`	1.14
`complab`	1.00		
`event`	1.19	`++`, `take`	1.44
`fft`	1.00	standard list functions	1.28
`genfft`	1.00	standard list functions	1.16
`ida`	1.16		
`listcompr`	0.84	`concat`, `++`	1.18
`parstof`	1.19		
`sched`	1.00		
`solid`	1.01	`foldl`, combined `area.icl` and `Csg.icl`	1.17
`transform`	1.02	standard list functions	1.03
`typecheck`	1.11	`++`, `map`, `concat`, `foldr`, `zip2`	1.30
`wang`	1.00	standard list functions	1.04
`wave4`	1.40		
`mergesort`	1.91		
`nfib`	6.73		
`generic conversion`	71.00		
`compiler`	1.05		
`compiler-monads`	1.25		

Fusion not only influences execution speed but also memory allocation. It appears that the decrease in memory usage is roughly twice as much as the decrease in execution time. For instance, the compiler itself runs approximately 5 percent faster whereas 9 percent less memory was allocated relative to the non-fused compiler. More or less the same holds for other programs.

Compilation with fusion enabled takes longer than without. Currently the difference is about 20 percent, when the implementation stabilises we expect to improve on this.

In the most expensive module that uses a monadic style only 68 percent of the curried function applications were eliminated. This improved the execution speed 33 percent and the memory allocation 51 percent. It should however be possible to remove nearly all these curried function applications. The current algorithm is not able to do this, because some higher-order functions are not optimized because they are indicated as accumulating. This is illustrated in the following example:

```
f :: [Int] Int -> Int
```

```
f [] s = s
f [e:l] s = g (add e) l s
    where add a b = a + b
g :: (Int -> Int) [Int] Int -> Int
g h l s = f l (h s)
```

The argument `h` of the function `g` is accumulating because `g` is called with `(add e)` as argument, therefore `g` is not fused with `(add e)`. In this case it would be safe to fuse. This limitation prevents the compiler from removing nearly all the remaining curried function applications from the above mentioned module. However, if a call `f (g x)`, for some functions `f` and `g`, appears in a program, the argument of the function `f` does not always have to be treated as an accumulating argument. This is the case when the argument of `g` is always the same or does not grow in the recursion. By recognizing such cases we hope to optimize most of the remaining curried function applications. Or instead, we could fuse a limited number of times in these cases, to make the fusion algorithm terminate.

Another example of curried applications in this module that cannot be optimized are `foldl` calls that yield a higher order function. Such a higher order function occurs at an accumulating argument position in the `foldl` call, and can therefore not be fused.

7 Related Work

Gill, Launchbury, and Peyton Jones [Gil93] use a restrictive consumer producer model by translating list functions into combinations of the primitive functions fold (consumer) and build (producer). This idea has been generalized to arbitrary data structures by Fegaras, Sheard and Zhou [Feg94], and also by Takano and Meijer [Tak95]. The approach of the latter is based on the category theoretical notion of *hylomorphism*. These hylomorphisms are the building blocks for functions. By applying transformation rules one can fuse these hylomorphisms resulting in deforested functions. These methods are able to optimize programs that cannot be improved by traditional deforestation. In particular, programs that contain reverse-like producers, i.e. producer functions with accumulators as arguments. On the other hand, Gill ([Gil96]) also shows some examples of functions that are deforested by the traditional method and not by these techniques. However, the main problem with these approaches is that they require that functions are written in some fixed format. Although for some functions this format can be generated from their ordinary definitions it is unclear how to do this automatically in general.

Peyton Jones and Marlow give a solid overview of the issues involved in transforming lazy functional programs in their paper in the related area of inlining [Pey99]. Specifically they identify code duplication, work duplication, and the uncovering of new transformation opportunities as three key issues to take into account.

Seidl and Sørensen [Sei97] develop a constraint-based system in an attempt to avoid the restrictions imposed by the purely syntactical approach used in the treeless approach to deforestation as used by Wadler [Wad88] and Marlow[Mar95]. Their analysis is a kind of abstract interpretation with which deforestation is approximated. This approximation results in a number of conditions on subterms and variables appearing in the program/function. If these conditions are met, it is guaranteed that deforestation will terminate. For instance, by using this more refined method the example program at the end of section 6.2 would be indicated as being safe.

Deforestation is also implemented in the compiler for the logic/functional programming language Mercury. To ensure termination of the algorithm a stack of unfolded calls is maintained, recursive calls can be unfolded only when they are smaller than the elements on the stack. This ordering is based on the sizes of the instantation tree of the arguments of a call. Accumulating parameters are removed from this sum of sizes. For details see [Tay98]. Our fusion algorithm can optimize some programs which the Mercury compiler does not optimize, for example ReverseFlatten from section 6.1

8 Conclusion

The original fusion algorithm has been extended and now combines deforestation together with dictionary elimination and higher-order removal. This adjusted algorithm has been implemented in the Clean 2.0 compiler allowing for tests on real-world applications. Initial results indicate that the main benefits are achieved for specialised features such as type classes, generics, and monads rather than in 'ordinary' code.

Further work remains to be done in the handling of accumulating parameters. Marlow presents a higher-order deforestation algorithm in his PhD thesis [Mar95] which builds on Wadler's original first-order deforestation scheme. A full comparison with the algorithm presented here remains to be done. Finally a formal proof of termination would be reassuring to have.

References

[Ach02] P. Achten, A. Alimarine, R. Plasmeijer. *When Generic Functions Use Dynamic Values.* Post-workshop submission: 14th International Workshop on the Implementation of Functional Languages, IFL 2002, Madrid, Spain, September 2002.

[Ali01] A. Alimarine and R. Plasmeijer. *A Generic Programming Extension for Clean.* In: Arts, Th., Mohnen M., eds. Proceedings of the 13th International Workshop on the Implementation of Functional Languages, IFL 2001, Selected Papers, Älvsjö, Sweden, September 24–26, 2001, Springer-Verlag, LNCS 2312, pages 168–185.

[Chin94] W.-N. Chin. *Safe fusion of functional expressions II: Further improvements* Journal of Functional Programming, Volume 6, Part 4, pp 515–557, 1994.

[Cle13] R. Plasmeijer, M. van Eekelen. *Language Report Concurrent Clean. Version 1.3.* Technical Report CSI R9816, NIII, University of Nijmegen, June 1998. Also available at `www.cs.kun.nl/~clean/contents/contents.html`

[Cle20] R. Plasmeijer, M. van Eekelen. *Language Report Concurrent Clean. Version 2.0. DRAFT!*, NIII, University of Nijmegen, December 2001. Also available at `www.cs.kun.nl/~clean/contents/contents.html`

[Feg94] L. Fegaras, T. Sheard, and T. Zhou. *Improving Programs which Recurse over Multiple Inductive Structures.* In Proc. of ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, Orlando, FL, USA, June 1994

[Fer88] A. Ferguson and P. Wadler. *When will Deforestation Stop.* In Proc. of 1988 Glasgow Workshop on Functional Programming, pp 39–56, Rothasay, Isle of Bute, August 1988.

[Fok95] J. Fokker. *Functional Specification of the JPEG algorithm, and an Implementation for Free*, In Programming Paradigms in Graphics, Proceedings of the Eurographics workshop in Maastricht, the Netherlands, september 1995, Wien, Springer 1995, pp102–120.

[Gil96] A. Gill. *Cheap Deforestation for Non-strict Functional Languages*,PhD Thesis, Department of Computing Science, Glasgow University, 1996.

[Gil93] A. Gill, S. Peyton Jones, J. Launchbury. *A Short Cut to Deforestation*, Proc. Functional Programming Languages and Computer Architecture (FPCA'93), Copenhagen, June 1993, pp223–232.

[Har93] P. Hartel, K. Langendoen. *Benchmarking Implementations of Lazy Functional Languages*, In Proc. of Functional Programming Languages and Computer Architecture, 1993, pp341–349.

[Har94] P. Hartel, *et al. Pseudoknot: A Float-Intensive Benchmark for Functional Compilers*, 6th Implementation of Functional Languages, School of Information Systems, University of East Anglia, Norwich, UK, 1994, pp13.1–13.34.

[Has92] P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, K. Hammond, J. Hughes, Th. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, J. Peterson. *Report on the programming language Haskell*, In ACM SigPlan Notices, 27 (5): 1–164. May 1992.

[Hin00] R. Hinze and S. Peyton Jones. *Derivable Type Classes.* In Graham Hutton, editor, Proceedings of the Fourth Haskell Workshop, Canada, 2000.

[Mar95] S. Marlow.*Deforestation for Higher-Order Functional Programs* PhD Thesis, Department of Computer Science, University of Glasgow, 1995.

[Pey99] S. Peyton Jones, S. Marlow. *Secrets of the Glasgow Haskell Compiler inliner* Workshop on Implementing Declarative Languages, 1999.

[Oka98] C. Okasaki. *Purely Functional Data Structures* Cambridge University Press, ISBN 0-521-63124-6, 1998.

[Sei97] H. Seidl, M.H. Sørensen. *Constraints to Stop Higher-Order Deforestation* In 24th ACM Symp. on Principles of Programming Languages, pages 400–413, 1997.

[Tak95] A. Takano, E. Meijer. *Shortcut to Deforestation in Calculational form*, Proc. Functional Programming Languages and Computer Architecture (FPCA'95), La Jolla, June 1995, pp 306–313.

[Tay98] S. Taylor.*Optimization of Mercury programs* Honours report, Department of Computer Science, University of Melbourne, Australia , November 1998.

[Wad88] P. Wadler. *Deforestation: Transforming Programs to Eliminate Trees* Proceedings of the 2nd European Symposium on Programming, Nancy, France, March 1988. Lecture Notes in Computer Science 300.

[Wad89] P. Wadler, S. Blott. *How to make ad-hoc polymorphism less ad hoc* In Proceedings 16th ACM Symposium on Principles of Programming Languages, pages 60–76, 1989.