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 paper describes the combined method that has been implemented in Clean 2.0 to perform various optimizations. This method is based on Chin's *fusion algorithm* [Chin94], which in its turn was inspired by Wadler's *deforestation algorithm* [Wad88,Fer88]. The two main differences between our method

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

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 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

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?"

$$\mathsf{append}(\mathsf{append}(a, b), c) \quad \text{or} \quad \mathsf{append}(a, \mathsf{append}(b, c))$$

where append is defined as

$$\begin{array}{lll} \mathsf{append}(f, t) = \mathsf{case} & f \ \mathsf{of} & \\ & \mathsf{Nil} & \mid t \\ & \mathsf{Cons}(x, xs) & \mid \mathsf{Cons}(x, \mathsf{append}(xs, t)) \end{array}$$

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 $\mathsf{app\_app}(a, b, c)$ and introduce a new function in which the two applications of append are combined:

$$\begin{array}{lll} \mathsf{app\_app}(a, b, c) = \mathsf{case} & a \ \mathsf{of} & \\ & \mathsf{Nil} & \mid \mathsf{append}(b, c) \\ & \mathsf{Cons}(x, xs) & \mid \mathsf{Cons}(x, \mathsf{app\_app}(xs, b, c)) \end{array}$$

Transforming the right expression introduces essentially the same function 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 are generally considered to be one of the most powerful concepts of functional languages. Firstly, type classes allow the language designer to support its implementation with a collection of libraries in which it is no longer necessary to use different names for conceptually the same operations. Secondly, it eases the definition of auxiliary operations on complex data structures. Both advantages are illustrated with 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* 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 `==` 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 realxed for dictionaries.

2.3 Higher-order function removal

Although dictionary elimination can be seen as a form of higher-order function removal, we describe these optimizations separately. 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 6.3.

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. Consider the following function definitions:

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

f g = f (twice g)
```

The parameter of `f` is accumulating: the argument 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 an 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. It appears that the combined data and higher-order fusion is sufficient to get rid of all intermediate data and all higher-order calls leading to specialized operations that are 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 $F + G(\ldots, \vec{E}, \ldots)$, by performing a sequence of unfold steps of both F and G. 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. 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 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(l) = case  l of
                   Nil          | Nil
                   Cons(x, xs)  | append(x, flatten(xs))
```

According to Chin, the append function is consuming in both of is 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.

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

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:

$$\begin{aligned} \mathsf{f}(x) &= \mathsf{g}(x) \\ \mathsf{g}(x) &= \mathsf{f}(\mathsf{h}(x)) \end{aligned}$$

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, 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 will take place. 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 will focus on program transformations in functional programming languages by describing a 'core language' capturing the essential aspects such as pattern matching, sharing and higher-order functions.[2]

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 @.

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).

[2] We leave out the treatment of dictionaries because it very much resembles the way other constructs are analyzed and transformed.

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.*

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

(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 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 ($x\ @ \ldots$), or x is used as an argument on an active position of a function. The idea is to define the notion 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.*

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

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* is borrowed from [Chin94] and slightly modified.

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 active and accumulating leads to the notion of consuming.

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* is also taken from [Chin94] again with a minor adjustment to deal with constructors.

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.*

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 notion of *linearity* is identical to that used by Chin.

Definition 6. *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 $F{+_i}S$ 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 three 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.
2. S is a partially applied function or F is a f-type function: the resulting function consists of the unfolding of F with the application of S substituted for formal argument i.
3. S is a constructor (either fully or partially applied): the resulting function consists of the unfolding of F with the application of S substituted for formal argument i.

Definition 7. *Rules for introducing new functions.*

$$
\begin{array}{lll}
\multicolumn{3}{l}{F{+_i}G(x_1,\dots,x_{i-1},y_1,\dots,y_m,x_{i+1},\dots,x_n)} \\
\quad & = \mathcal{T}[\![E[E'/x_i]]\!], & \textit{if } \mathsf{arity}(G) = m \textit{ and } F \textit{ is a g-type function} \\
 & & \textit{where } G\vec{y} = E' \\
 & = \mathcal{T}[\![E[G(y_1,\dots,y_m)/x_i]]\!], & \textit{otherwise} \\
\multicolumn{3}{l}{F{+_i}C(x_1,\dots,x_{i-1},y_1,\dots,y_m,x_{i+1},\dots,x_n)} \\
 & = \mathcal{T}[\![E[C(y_1,\dots,y_m)/x_i]]\!] & \\
 & & \\
\multicolumn{3}{l}{F^{+}(x_1,\dots,x_n,x)} \\
 & = \mathcal{T}[\![\mathcal{R}_x[\![E]\!]]\!] & \\
 & & \\
\multicolumn{3}{l}{\mathcal{R}_y[\![x]\!]} \\
 & = x \mathbin{@} y & \\
\multicolumn{3}{l}{\mathcal{R}_y[\![F(E_1,\dots,E_n)]\!]} \\
 & = F(E_1,\dots,E_n,y), & \textit{if } \mathsf{arity}(F) > n \\
 & = F(E_1,\dots,E_n) \mathbin{@} y, & \textit{otherwise} \\
\multicolumn{3}{l}{\mathcal{R}_y[\![\mathsf{let}\ \vec{x} = \vec{E}\ \mathsf{in}\ E']\!]} \\
 & = \mathsf{let}\ \vec{x} = \vec{E}\ \mathsf{in}\ \mathcal{R}_y[\![E']\!] & \\
\multicolumn{3}{l}{\mathcal{R}_y[\![\mathsf{case}\ E\ \mathsf{of}\ \dots P_i | E_i \dots]\!]} \\
 & = \mathsf{case}\ E\ \mathsf{of}\ \dots P_i | \mathcal{R}_y[\![E_i]\!] \dots & \\
\multicolumn{3}{l}{\mathcal{R}_y[\![E \mathbin{@} E']\!]} \\
 & = (E \mathbin{@} E') \mathbin{@} y &
\end{array}
$$

Note that we generate each $F{+_i}S$ only once. We use F^+ as a name for the function that results from raising the arity of F by one.

Definition 8. *Transformation rules for first and higher-order expressions*

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

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{F}\llbracket F(E_1, \ldots, E_i, \ldots, E_m) \rrbracket} \\
 & = \mathcal{F}\llbracket F{+_i}S(E_1, \ldots, E_{i-1}, E'_1, \ldots, E'_n, E_{i+1} \ldots, E_m) \rrbracket, \\
 & \quad \textit{if: for some } i \textit{ with } E_i = S(E'_1, \ldots, E'_n) \\
 & \quad \textit{1) } F \textit{ is consuming in } i \\
 & \quad \textit{2) } S(E'_1, \ldots, E'_n) \textit{ is a producer} \\
 & \quad \textit{3) } F \textit{ is linear in } i \textit{ if } \mathsf{arity}(S) = n \\
 & \quad \textit{or} \\
 & \quad \textit{1) } F \textit{ is both consuming and linear in } i \\
 & \quad \textit{2) } \mathsf{arity}(F) = m \textit{ and } \mathsf{arity}(S) = n \\
 & \quad \textit{3) } S(E'_1, \ldots, E'_n) \textit{ has a higher order type} \\
 & = F(E_1, \ldots, E_i, \ldots, E_m), \textit{ otherwise}
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{H}\llbracket C(E_1, \ldots, E_k)\ \texttt{@}\ E \rrbracket & = C(E_1, \ldots, E_k, E) \\
\mathcal{H}\llbracket F(E_1, \ldots, E_k)\ \texttt{@}\ E \rrbracket & = \mathcal{F}\llbracket F(E_1, \ldots, E_k, E) \rrbracket, \textit{ if } \mathsf{arity}(F) > k \\
 & = \mathcal{F}\llbracket F^+(E_1, \ldots, E_k, E) \rrbracket, \textit{ otherwise}
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{C}\llbracket \mathsf{case}_F G(E_1, \ldots, E_n)\ \mathsf{of}\ \ldots \rrbracket} \\
 & = \mathcal{F}\llbracket F(x_1, \ldots, x_{i-1}, G(E_1, \ldots, E_n), x_{i+1}, \ldots, x_n) \rrbracket \\
 & \quad \textit{where} \\
 & \qquad\qquad F(x_1, \ldots, x_n) = \mathsf{case}\ x_i\ \mathsf{of}\ \ldots \\
\multicolumn{2}{l}{\mathcal{C}\llbracket \mathsf{case}_F C_i(E_1, \ldots, E_n)\ \mathsf{of}\ \ldots C_i(x_1, \ldots, x_n) | E'_i \ldots \rrbracket} \\
 & = \mathcal{T}\llbracket E'_i[E_1/x_1, \ldots, E_n/x_n] \rrbracket \\
\multicolumn{2}{l}{\mathcal{C}\llbracket \mathsf{case}_F (\mathsf{case}\ E\ \mathsf{of}\ \ldots P_i | E'_i \ldots)\ \mathsf{of}\ \ldots \rrbracket} \\
 & = \mathsf{case}\ \mathcal{T}\llbracket E \rrbracket\ \mathsf{of}\ \ldots P_i | E''_i \ldots \\
 & \quad \textit{where } E''_i = \mathcal{F}\llbracket F(x_1, \ldots, x_{i-1}, \mathcal{T}\llbracket E'_i \rrbracket, x_{i+1}, \ldots, x_n) \rrbracket^3 \\
 & \quad \textit{and} \\
 & \qquad\qquad F(x_1, \ldots, x_n) = \mathsf{case}\ x_i\ \mathsf{of}\ \ldots \\
\mathcal{C}\llbracket \mathsf{case}_F x\ \mathsf{of}\ \ldots P_i | E_i \ldots \rrbracket & = \mathsf{case}_F x\ \mathsf{of}\ \ldots P_i | \mathcal{T}\llbracket E_i \rrbracket \ldots
\end{array}
$$

[3] A minor improvement here is obtained by examining the expression E'_i. If this expression starts with a constructor, it is better to perform the pattern match instead of undoing the unfold step of F.

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, $\mathsf{Flatten}$ and $\mathsf{Reverse}$. The first two functions have been defined earlier. We will use three different versions of $\mathsf{Reverse}$ to illustrate the effect of alternative, but equivalent, function definitions on the fusion result. The applications being fused are the right-hand sides of the functions called $\mathsf{test1}$, $\mathsf{test2}$, and $\mathsf{test3}$.

The first definition of `Reverse` uses a helper function with an explicit accumulator:

$$
\begin{array}{lll}
\mathsf{Reverse1}(l) & = \mathsf{Rev}(l, \mathsf{Nil}) & \\
 & & \\
\mathsf{Rev}(l, a) & = \mathsf{case}\ l\ \mathsf{of} & \\
 & \quad \mathsf{Nil} & \mid a \\
 & \quad \mathsf{Cons}(x, xs) & \mid \mathsf{Rev}(xs, \mathsf{Cons}(x, a)) \\
 & & \\
\mathsf{test1}(l) & = \mathsf{Reverse1}(\mathsf{Flatten}(l)) &
\end{array}
$$

The result of applying the transformation rules to the function $\mathsf{test1}$ is shown below. To make the description less baroque we have left out the + and subscript in the names of the combined functions.

$$
\begin{array}{lll}
\mathsf{test1}(l) & = \mathsf{Reverse1Flatten}(l) & \\
 & & \\
\mathsf{Reverse1Flatten}(l) & = \mathsf{RevFlatten}(l, \mathsf{Nil}) & \\
 & & \\
\mathsf{RevFlatten}(l, r) & = \mathsf{case}\ l\ \mathsf{of} & \\
 & \quad \mathsf{Nil} & \mid r \\
 & \quad \mathsf{Cons}(x, xs) & \mid \mathsf{RevAppendFlatten}(x, xs, r) \\
 & & \\
\multicolumn{3}{l}{\mathsf{RevAppendFlatten}(xs, l, r)} \\
 & = \mathsf{case}\ xs\ \mathsf{of} & \\
 & \quad \mathsf{Nil} & \mid \mathsf{RevFlatten}(l, r) \\
 & \quad \mathsf{Cons}(x, xs) & \mid \mathsf{RevAppendFlatten}(xs, l, \mathsf{Cons}(x, r))
\end{array}
$$

The transformation steps that were needed to achieve part of this result are:

$$
\begin{array}{l}
\mathcal{T}[\![\mathsf{Reverse1}(\mathsf{Flatten}(l))]\!] \\
\quad = \mathcal{F}[\![\mathsf{Reverse1}(\mathsf{Flatten}(l))]\!] \\
\quad = \mathsf{Reverse1Flatten}(l) \\
\mathsf{Reverse1Flatten}(l) \\
\quad = \mathcal{T}[\![\mathsf{Rev}(\mathsf{Flatten}(l), \mathsf{Nil})]\!] \\
\quad = \mathsf{RevFlatten}(l, \mathsf{Nil})
\end{array}
$$

$$\begin{array}{l}
\mathsf{RevFlatten}(l, r) \\
\quad = \mathcal{C}[\![\mathsf{case}\ \left(\begin{array}{lll}\mathsf{case} & l\ \mathsf{of} & \\ & \mathsf{Nil} & \mid \mathsf{Nil} \\ & \mathsf{Cons}(x, xs) & \mid \mathsf{Append}(x, \mathsf{Flatten}(xs))\end{array}\right)\ \mathsf{of} \\
\qquad\qquad \begin{array}{ll}\mathsf{Nil} & \mid r \\ \mathsf{Cons}(x, xs) & \mid \mathsf{Rev}(xs, \mathsf{Cons}(x, r))]\!]\end{array} \\
\quad = \mathsf{case}\ \ l\ \mathsf{of} \\
\qquad\quad \begin{array}{ll}\mathsf{Nil} & \mid r^4 \\ \mathsf{Cons}(x, xs) & \mid \mathcal{F}[\![\mathsf{Rev}(\mathsf{Append}(x, \mathsf{Flatten}(xs)), r)]\!]\end{array} \\
\quad = \mathsf{case}\ \ l\ \mathsf{of} \\
\qquad\quad \begin{array}{ll}\mathsf{Nil} & \mid r \\ \mathsf{Cons}(x, xs) & \mid \mathcal{F}[\![\mathsf{RevAppend}(x, \mathsf{Flatten}(xs)), r)]\!]\end{array} \\
\quad = \mathsf{case}\ \ l\ \mathsf{of} \\
\qquad\quad \begin{array}{ll}\mathsf{Nil} & \mid r \\ \mathsf{Cons}(x, xs) & \mid \mathcal{F}[\![\mathsf{RevAppendFlatten}(x, xs, r)]\!]\end{array}
\end{array}$$

The generation of RevAppendFlatten out of RevAppend and Flatten proceeds in the same way.

The second definition of Reverse uses the standard higher-order *foldl* function:

$$\begin{array}{lll}
\mathsf{Reverse2}(l) & = \mathsf{Foldl}(\mathsf{Snoc}, \mathsf{Nil}, l) & \\
& & \\
\mathsf{Foldl}(f, r, l) & = \mathsf{case}\ l\ \mathsf{of} & \\
& \quad \mathsf{Nil} & \mid r \\
& \quad \mathsf{Cons}(x, xs) & \mid \mathsf{Foldl}(f, (f\ @\ r)\ @\ x, xs) \\
& & \\
\mathsf{Snoc}(xs, x) & = \mathsf{Cons}(x, xs) & \\
& & \\
\mathsf{test2}(l) & = \mathsf{Reverse2}(\mathsf{Flatten}(l)) &
\end{array}$$

Transforming test2 results in two mutually recursive functions that are, except for the order of the parameters, identical to the auxiliary functions RevFlatten and RevAppendFlatten generated by the transformation of test1.

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

$$\begin{array}{lll}
\mathsf{Reverse3}(l) & = \mathsf{case}\ l\ \mathsf{of} & \\
& \quad \mathsf{Nil} & \mid \mathsf{Nil} \\
& \quad \mathsf{Cons}(x, xs) & \mid \mathsf{Append}(\mathsf{Reverse3}(xs), \mathsf{Cons}(x, \mathsf{Nil})) \\
& & \\
\mathsf{test3}(l) & = \mathsf{Reverse3}(\mathsf{Flatten}(l)) &
\end{array}$$

When transforming test3 again Reverse3 will be combined with Flatten but the fact that Reverse3 itself is not a producer (the recursive occurrence of this function appears on a consuming position), will prevent the combined right-hand side from being deforested completely.

[4] Here we have applied the optimization step that was suggested in the footnote of definition 8.

In general combinations of standard list functions (except for *accumulating* functions, such as Reverse), e.g.

$$\mathsf{Sum}(\mathsf{Map\ Inc}(\mathsf{Map\ Inc}(\mathsf{Take}(n, \mathsf{Repeat}(1)))))$$

are transformed into a single function that generates no list at all and that does not contain any higher-order function applications.

6.2 Generics

Generic functions are translated into real functions in three steps. The first step, taking place before type checking, converts the generic specifications into class definitions and instances of those classes.[5] Furthermore, conversion functions (so-called *bidirectional mappings*) are generated to convert the function arguments and result to and from the generic representation.

The second translation step takes place during type checking: all overloading is resolved and replaced by appropriate dictionaries.

Finally, the overhead introduced by dictionaries, higher-order functions and by using the generic representations is removed by the fusion algorithm. For an in-depth description we refer to [Ali01]. Here, we briefly illustrate fusion with the aid of the example mentioned in the introduction. Suppose we have used the instance of equality for lists of integers, called `eqListInt`. This function can be directly expressed in terms of the library function `eqInt`, for comparing two integers, and the function `eqList`, for comparing lists. The latter is generated according to the generic specification of equality. The result is shown below:

```
eqListInt l1 l2 = eqList eqInt l1 l2

eqList eqa x y = fromBM (bimapEq isoList) (eqListG eqa) x y
```

The bidirectional mapping `bimapEq isoList` converts a function of type `[a] -> [a] -> Bool` to and from a function of type `GList a -> GList a -> Bool`, where `GList` denotes the generic list type. In this example we need the 'from' version selected by `fromBM`. The function `eqListG` is defined directly in terms of the user defined instances of equality for the generic type constructors. The structure of the list type itself determines how these user defined instances are combined.

```
eqListG eqa = eqEITHER eqUNIT (eqPAIR eqa (eqList eqa))
```

The complete program contains nearly twenty auxiliary operations that are used to define `bimapEq` and `isoList`. There is no need to say that, without any further optimizations, operations defined in this way are very inefficient. It is pointless to show all fusion steps that were performed in order to obtain the optimized

[5] Currently the programmer has to specify for which types an instance is wanted. We could also generate instances lazily, i.e. when the compiler detects that a certain instance is required. For technical reasons this appears to be difficult.

result: it appears that no less than 70 intermediate functions were generated. However, most of these become obsolete after the fusion process. After removing these superfluous functions we end up with the following result where the names of the generated functions give some information about the functions that were combined.

```
eqListeqInt x y =
    eqEITHEReqUNITeqPAIReqListeqInteqIntmapListTomapListTo x y

eqEITHEReqUNITeqPAIReqListeqInteqIntmapListTomapListTo l1 l2 =
    case l1 of
        Nil = eqEITHERLEFTeqUNITUNITmapListTo l2
        Cons x xs
            = eqEITHERRIGHTeqPAIReqListeqInteqIntPAIRmapListTo x xs l2

eqEITHERLEFTeqUNITUNITmapListTo l =
    case l of
        Nil = True
        Cons x xs
            = False

eqEITHERRIGHTeqPAIReqListeqInteqIntPAIRmapListTo y ys l =
    case l of
        Nil = False
        Cons x xs
            = eqPAIR2eqListeqInteqIntPAIR y ys x xs

eqPAIR2eqListeqInteqIntPAIR y ys x xs =
    And (Equal x y) (eqListeqInt xs ys)
```

This result is essentially the same as any efficient hand written program.

6.3 Self-application

As a practical test of the effectiveness of the adjusted fusion algorithm we compared the memory allocation and speed of a Clean 2.0 compiler generated with fusion enabled with that of the same compiler generated without fusion. In both cases specialization of overloaded functions (i.e. dictionary removal) was enabled.

The fused compiler compiled its own code approximately 5 percent faster, and 10 percent less memory was allocated relative to the non-fused compiler. These 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. Note that generics are not used inside the compiler.

In one of these modules all curried function applications introduced by the monadic style were eliminated. Due to these optimizations the module executed 25 percent faster and allocated 50 percent less memory.

However in the most expensive module using a monadic style only 70 percent of the curried function applications were eliminated. This improved the execution speed 30 percent and the memory allocation 50 percent.

It should be possible to remove nearly all these curried function applications. The current algorithm is not able to do this, because some functions that are called with a higher order function as argument are not optimized, because the argument is accumulating.

This happens also in the following example:

```
add :: Int Int -> Int
add a b = a+b

f :: [Int] Int -> Int
f [e:l] s = g (add e) l s
f [] s = s

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 and Marlow. 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 6.3 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 Reverse1Flatten 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

[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, Faculty of mathematics and Informatics, Catholic 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!* Faculty of mathematics and Informatics, Catholic 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.

[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.

[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, Montreal, Canada, September 17, 2000.

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