A Short Cut to Deforestation

Andrew Gill John Launchbury Simon L Peyton Jones
Department of Computing Science, University of Glasgow G12 8QQ
{andy,jl,simonpj}@dcs.glasgow.ac.uk

This paper is to appear in FPCA 1993.

Abstract

Lists are often used as "glue" to connect separate parts of a program together. We propose an automatic technique for improving the efficiency of such programs, by removing many of these intermediate lists, based on a single, simple, local transformation. We have implemented the method in the Glasgow Haskell compiler.

1 Introduction

Functional programs are often constructed by combining together smaller programs, using an intermediate list to communicate between the pieces. For example, the function `all`, which tests whether all the elements of a list `xs` satisfy a given predicate `p` may be defined as follows (we use the language *Haskell* throughout this paper (Hudak et al. [1992])):

```
all p xs = and (map p xs)
```

Here, `p` is applied to all the elements of the list `xs`, producing an intermediate list of booleans. These booleans are "anded" together by the function `and`, producing a single boolean result. The intermediate list is discarded, and eventually recovered by the garbage collector.

This compositional style of programming is one reason why lists tend to be so pervasive, despite the availability of user-defined types. Functional languages support this tendency by supplying a large library of pre-defined list-manipulating functions, and by supporting special syntax for lists, such as list comprehensions. Then, because so many functions are already available to manipulate lists, it is easy to define new functions which work on lists, and so on.

Unfortunately, all these intermediate lists give rise to an efficiency problem. Each of their *cons* cells has to be allocated, filled, taken apart, and finally deallocated, all of which consumes resources. There are more efficient versions of `all` which do not use intermediate lists. For example, it can be re-written like this:

```
all' p xs = h xs
  where h []     = True
        h (x:xs) = p x && h xs
```

Now no intermediate list is used, but this has been achieved at the cost of clarity and conciseness compared with the original definition.

We want to eat our cake and have it too. That is, we would like to write our programs in the style of `all`, but have the compiler automatically transform this into the more efficient version `all'`.

One example of just such a transformation is *deforestation* (Wadler [1990]). Deforestation removes arbitrary intermediate data structures (including lists), but suffers from some major drawbacks (Section 2). As a result, apart from a prototype incorporated into a local version of the Chalmers LML compiler (Davis [1987]), we know of no mature compiler that uses deforestation as part of its regular optimisations.

In this paper we present a cheap and easy way of eliminating many intermediate lists that does not suffer from these drawbacks. Our optimisation scheme is based on a single, simple, local transformation, and is practical for inclusion in a real compiler. It has the following characteristics:

- The technique applies to most of the standard list processing functions. Examples are functions that consume lists, such as `and` and `sum`, expressions that create lists, such as `[x..y]`, and functions that both consume and create lists, such as `map`, `filter`, `++` and the like. In general, the technique handles any *compositional* list-consuming function, that is, one which can be written using `foldr`.
- The technique extends straightforwardly to improve the state of the art in compiling list comprehensions (Section 4). Standard compilation techniques for list comprehensions build an intermediate list in expressions such as
  ```
  [ f x | x <- map g xs, odd x ]
  ```
  Our technique automatically transforms this expression to a form that uses no intermediate lists. Furthermore, we are able to use the same technology to

avoid intermediate lists in Haskell's array comprehensions (Section 5).

- Like standard deforestation, our method can be applied to other data structures, such as trees and other more complicated datatypes, though we have not implemented this generalisation so far.

The effects can be startling. For example, in one program which makes heavy use of list comprehensions (the 8-queens program), we have measured a three-fold speedup as a result of applying our technique. While this is probably an upper bound on the expected improvement in larger programs, the technique is very cheap to implement, so it provides reasonable gains at very low cost.

2 Deforestation

The core of Wadler's deforestation algorithm consists of seven transformation rules, four of which focus on terms built using the `case` construct. Of these, apart from the rule unfolding function calls, the only rule which actually removes computation is the `case`-on-constructor rule. Using this rule, the term,

```
case (C2 t1 t2 t3) of
  C1 x1 x2    -> e1
  C2 x1 x2 x3 -> e2
  C3 x1       -> e3
```

is translated to

```
let x1 = t1
    x2 = t2
    x3 = t3
in
  e2
```

The heap object represented by the constructor `C2` is never built nor examined, so saving computation. Indeed, this transformation is precisely where the intermediate data structure is eliminated, and the purpose of all the other transformation rules is to provide opportunities for applying this key one.

There is a complication, however. When recursive functions are involved there is a risk of performing infinite transformation by repeatedly unfolding the function definition. To alleviate this, the algorithm has to keep track of which function calls have occurred previously, and so generate a suitable recursive definition when a repeat occurs. This is a *fold* step in Burstall and Darlington's sense (Burstall & Darlington [1977]). Keeping track of these function calls and constantly checking for repeats introduces substantial cost and complexity into the algorithm. What is worse, in general it is not foolproof. Given arbitrary programs, it is possible that the patterns of function calls never repeat themselves. This leads to infinite unfolding, and consequent non-termination of the compiler.

Turchin addressed this problem in his *supercompiler* (which performed a deforestation-like transformation) by using a generalisation phase (Turchin [1988]). Function calls are squeezed into fairly restricted patterns which the user has to decide beforehand.

```
foldr k z []     = z
foldr k z (x:xs) = k x (foldr k z xs)

and xs       = foldr (&&) True xs
sum xs       = foldr (+) 0 xs
elem x xs    = foldr (\ a b -> a == x || b)
                     False xs
map f xs     = foldr (\ a b -> f a : b) [] xs
filter f xs  = foldr (\ a b -> if f a
                               then a:b
                               else b)
                     [] xs
xs ++ ys     = foldr (:) ys xs
concat xs    = foldr (++) [] xs
foldl f z xs = foldr (\ b g a -> g (f a b))
                     id xs z
```

Figure 1: Examples of functions using `foldr`

Wadler's solution is different. He attacks the problem by limiting the form of the input program to compositions of functions in so-called *treeless form*. This is a particularly restrictive form of function definition which severely limits the general applicability of the algorithm. Not only is it first-order, but all variables must be used linearly, and no internal data structures are permitted. These restrictions did however allow two theorems to be proved: first that the result was also in treeless form (guaranteeing that *no* intermediate structures were built), and secondly that the algorithm does always terminate (Ferguson & Wadler [1988]).

There have been various attempts to lift some of Wadler's restrictions (Chin [1990]; Marlow & Wadler [1993]), but many problems remain.

3 Transformation

The approach we take here is less purist, but more practical. We do not guarantee to remove *all* intermediate structures (and indeed in a general program we could not hope to do so), but we do allow all legal programs as input. Furthermore, like Wadler, we do guarantee termination.

In Wadler's case, proving termination was far from trivial, as recursive functions were frequently unfolded. In our case, a termination proof is trivial as we do not explicitly unfold recursive functions. Rather, we obtain a similar effect by performing algebraic transformations on pre-defined functions.

As an example of how this may occur, here is a possible algebraic transformation which eliminates an intermediate list:

```
map f (map g xs) = map (f.g) xs
```

Unfortunately, we would need a huge set of rules to account for all possible pairs of functions. Our approach reduces this set to a single rule, by *standardising the way in which lists are consumed (Section 3.1), and standardising the way in which they are produced (Section 3.2).*

3.1 Consuming Lists

A function which consumes a list in a uniform fashion can always be expressed by replacing the *conses* in the list with a given function $\oplus$, and replacing the *nil* at the end of the list by a given value z. This operation is encapsulated by the higher-order function `foldr`, which can be informally defined like this:

$$\texttt{foldr}\ (\oplus)\ z\ [x_1, x_2, \ldots, x_n] = x_1 \oplus (x_2 \oplus (\cdots (x_n \oplus z)))$$

The Haskell definition of `foldr` is given in Figure 1. Very many list-consuming functions are "uniform" in this sense, and can be expressed directly in terms of `foldr`, and some of these are also given in Figure 1. The first three functions, `and`, `sum` and `elem`, are purely list consumers (`elem` is a list-membership test). The next few both consume and produce lists. Finally, even `foldl`, which consumes a list in a left-associative way, can be defined in terms of `foldr`.

One tempting method of exploiting this regularity is to express programs using `foldr`, and then use a small set of transformation rules on `foldr`. However this scheme fails because there is no general rule to transform a composition of `foldr` with itself. For example consider:

```
and (map f xs)
```

We can begin by unfolding `and` and `map` to their `foldr` equivalents:

```
foldr (&&) True (foldr ((:).f) [] xs)
```

But now we are stuck because of the lack of `foldr/foldr` rule.

$$\texttt{foldr}\ k_1\ z_1\ (\texttt{foldr}\ k_2\ z_2\ e) = \ ???$$

It is certainly possible to invent a more specialised transformation, such as this one:

$$\texttt{foldr}\ k_1\ z_1\ (\texttt{foldr}\ ((:).k_2)\ []\ e) = \texttt{foldr}\ (k_1.k_2)\ z_1\ e$$

This will successfully transform our example to:

```
foldr ((&&).f) True xs
```

But now we have run straight back into the problem of an explosion in the number of rules. How do we know when we have "enough" rules? How can we be sure we do not have "redundant" rules?

The reason that there is no `foldr/foldr` rule is because the outer `foldr` has no "handle" on the way in which the inner `foldr` is producing its output list. We need a way of identifying exactly where in a term the *cons* cells of an output list are being produced: in effect, we need the dual to `foldr`.

3.2 Producing Lists

The effect of the application `foldr k z xs` is to replace each *cons* in the list `xs` with `k` and to replace the *nil* in `xs` with `z`. So if we abstract list-producing functions with respect to *cons* and *nil*, we can obtain the effect of the `foldr` merely by applying the abstracted list-producing function to `k` and `z`. Equivalently, if we define a function `build` like this:

```
build g = g (:) []
```

then we may hope that, for all g, k and z,

$$\boxed{\texttt{foldr}\ k\ z\ (\texttt{build}\ g)\ =\ g\ k\ z}$$

We call this equivalence the `foldr/build` rule. To ensure its validity we need to impose some restrictions on g, but we will address that later (Section 3.4).

To take an example, consider the `from` function which when applied to two numbers produces a list of numbers, starting from the first and counting up to the second. Originally we may have defined `from` by

```
from a b = if a>b
           then []
           else a : from (a+1) b
```

but abstracting the definition over *cons* and *nil* gives,

```
from' a b = \ c n -> if a>b
                     then n
                     else c a (from' (a+1) b c n)
```

The original `from` can be obtained thus:

```
from a b = build (from' a b)
```

We can now deforest uses of this new version of `from` so long as its list is consumed using `foldr`. For example,

```
sum (from a b)
  = foldr (+) 0 (build (from' a b))
  = from' a b (+) 0
```

(We annotate instances of the `foldr/build` rule in all our examples using `foldr` ··· `build` to make the reduction explicit.) Now no intermediate list is produced. Deforestation has been successful.

In short, *provided we build lists using* `build` *and consume them using* `foldr`, *the* `foldr/build` *rule alone suffices to deforest compositions of such functions.* Figure 2 gives the definitions of a number of list-producing functions in terms of `build`. The definitions of functions which consume a list as well as producing one involve `foldr` as well as `build`. Functions which simply consume a list are defined solely in terms of `foldr`, as in Figure 1, and are not repeated here.

3.3 An example: `unlines`

As a slightly larger example of the `foldr/build` rule in action, consider the function `unlines`, taken from the Haskell prelude.

```
unlines ls = concat (map (\l -> l ++ ['\n']) ls)
```

This function takes a list of strings, and joins them together, inserting a newline character after each one. An intermediate version of the list of strings is created, together with an intermediate version of each string (when the newline character is appended).

To deforest this definition, we first unfold the standard functions `concat`, `map` and `++`, using the definitions in Figure 2, to give:

```
unlines ls = build
  (\c0 n0 -> foldr (\xs b -> foldr c0 b xs) n0 (build
    (\c1 n1 -> foldr (\l t -> c1 (build
      (\c2 n2 -> foldr c2 (foldr c2 n2 (build
        (\c3 n3 -> c3 '\n' n3))) l )) t) n1 ls )))
```

```
map f xs    = build (\ c n -> foldr (\ a b -> c (f a) b) n xs)
filter f xs = build (\ c n -> foldr (\ a b -> if f a then c a b else b) n xs)
xs ++ ys    = build (\ c n -> foldr c (foldr c n ys) xs)
concat xs   = build (\ c n -> foldr (\ x y -> foldr c y x) n xs)

repeat x    = build (\ c n -> let r = c x r in r)
zip xs ys   = build (\ c n -> let zip' (x:xs) (y:ys) = c (x,y) (zip' xs ys)
                                  zip' _      _      = n
                              in  zip' xs ys)
[]          = build (\ c n -> n)
x:xs        = build (\ c n -> c x (foldr c n xs))
```

Figure 2: Definitions of standard functions using `foldr` and `build`

Now we apply the `foldr/build` rule in the two places that are marked, to give:

```
unlines ls = build
  (\c0 n0 ->
    (\c1 n1 -> foldr (\l t -> c1 (build
      (\c2 n2 -> foldr c2 (
        (\c3 n3 -> c3 '\n' n3) c2 n2 ) l)) t) n1 ls)
        (\xs b -> foldr c0 b xs) n0)
```

Performing three β-reductions gives:

```
unlines ls = build
  (\ c0 n0 -> foldr (\l t -> foldr c0 t(build
    (\c2 n2 -> foldr c2 (c2 '\n' n2) l))) n0 ls)
```

This in turn exposes a new opportunity to use the `foldr / build` rule. After using it, and performing some more β-reductions we get:

```
unlines ls = build
 (\c0 n0 -> foldr (\l b -> foldr c0 (c0 '\n' b) l) n ls)
```

Now no more applications of the transformation are possible. We may choose to leave the definition in this form, so that any calls of `unlines` may also be deforested. Alternatively, we may now inline `build`, revealing the `(:)`'s and `[]`. After simplification we get,

```
unlines ls
  = foldr (\l b -> foldr (:) ('\n' : b) l) [ ] ls
```

If we also inline `foldr` we get,

```
unlines ls = h ls
  where h []     = []
        h (l:ls) = g l
          where g []     = '\n' : h ls
                g (x:xs) = x : g xs
```

This is as efficient a coding of `unlines` as we may reasonably hope for.

The example also makes it clearer how the technique works: the `foldr` at the input end of a list-consuming function "cancels out" the `build` at the output end of a list-producing one. This cancellation may in turn bring together a further `foldr/build` pair, and so on. A pipeline (composition) of list-transforming functions gives rise to a composition of `foldr`s and `build`s which can crudely be pictured like this:

```
... foldr] [build foldr] [build ...
```

(The square brackets indicate the related origin of the pair.) But now, by simply rebracketing we get the composition

```
... [foldr build] [foldr build] ...
```

and the inner `foldr/build` pairs now "cancel".

3.4 Correctness

There appears to be a serious problem with the approach we have described: the `foldr/build` rule is patently false! For example,

```
foldr k z (build (\c n -> [True]))
                  ≢
        (\c n -> [True]) k z
```

Using the definitions of `foldr` and `build` we can see that the left-hand side is `k True z`, while the right hand side is just `[True]`. These two values do not even necessarily have the same type!

In this counter-example the trouble is that the function passed to `build` constructs its result list without using `c` and `n`. The time we can guarantee that the `foldr/build` rule does hold is when g truly is a list which has been "uniformly" abstracted over all its *conses* and *nil*.

It turns out that we can guarantee this property simply by restricting g's type! Suppose that g has the type

$$g : \forall\beta.(A \to \beta \to \beta) \to \beta \to \beta$$

for some fixed type A. Then, informally, we can argue as follows. Because g is polymorphic in β, it can only manufacture its result (which has type β) by using its two arguments, k and z. Furthermore, the types of k and z mean that they can only be composed into an expression of the form

$$(k\ e_1\ (k\ e_2 \ldots (k\ e_n\ z) \ldots))$$

which is exactly the form we require.

This arm waving is obviously unsatisfactory. Rather delightfully, the theorem we want turns out to be a direct consequence of the "free theorem" for g's type (Reynolds [1983]; Wadler [1989]).

Theorem
If for some fixed A we have

$$g : \forall\beta.(A \to \beta \to \beta) \to \beta \to \beta$$

then

$$\texttt{foldr}\ k\ z\ (\texttt{build}\ g) = g\ k\ z$$

Proof
The "free theorem" associated with g's type is that, for all types B and B', and functions $f : A \to B \to B$, $f' : A \to B' \to B'$, and (a strict) $h : B \to B'$ the following implication holds:

$$\begin{array}{l}(\forall a : A.\ \forall b : B.\ h\ (f\ a\ b) = f'\ a\ (h\ b)) \\ \Rightarrow\ (\forall b : B.\ h\ (g_B\ f\ b) = g_{B'}\ f'\ (h\ b))\end{array}$$

where g_B and $g_{B'}$ are the instances of g at B and B' respectively. (From now on we will drop the subscripts from g as languages like Haskell have silent type instantiation).

We now instantiate this result. Let $h = \texttt{foldr}\ k\ z$, $f = (\texttt{:})$, and $f' = k$. Now the theorem says,

$$\begin{array}{l}(\forall a : A.\ \forall b : B.\ \texttt{foldr}\ k\ z\ (a : b) = k\ a\ (\texttt{foldr}\ k\ z\ b)) \\ \Rightarrow\ (\forall b : B.\ \texttt{foldr}\ k\ z\ (g\ (\texttt{:})\ b) = g\ k\ (\texttt{foldr}\ k\ z\ b))\end{array}$$

The left hand side is a consequence of the definition of `foldr`, so the right hand side follows. That is,

$$\forall b : B.\ \texttt{foldr}\ k\ z\ (g\ (\texttt{:})\ b) = g\ k\ (\texttt{foldr}\ k\ z\ b)$$

Now let $b = \texttt{[]}$. By definition, $\texttt{foldr}\ k\ z\ \texttt{[]} = z$, so finally we obtain,

$$\texttt{foldr}\ k\ z\ (g\ (\texttt{:})\ \texttt{[]}) = g\ k\ z$$

which, given the definition of `build`, is exactly what we require. □

The impact of this result is significant: so long as `build` is only applied to functions of the appropriate type, the deforestation transformations may proceed via the `foldr/build` rule with complete security. Furthermore, since all our program transformations preserve types, it is only necessary to check that the original introductions of `build` (in Figure 2) are correct.

The ideal way to proceed would be to define `build` with the type:

$$\texttt{build} : \forall\alpha.\ (\forall\beta.\ (\alpha \to \beta \to \beta) \to \beta \to \beta) \to \mathit{List}\ \alpha$$

and then have the compiler's type-checker confirm that all applications of `build` are well-typed. Unfortunately, Haskell's type system is based on the Hindley-Milner system (Milner [1978]), which does not allow local quantification; that is, the $\forall$'s must all be at the top level of a type. (A more general type system, such as that of Ponder (Fairbairn [1985]) or Quest (Cardelli & Longo [1991]) would allow this type for `build`, but they lack the type-inference property.)

We sidestep this by building into the compiler trusted definitions of all the standard functions (`map`, `filter`, `concat` and so on) in terms of `build` and `foldr`. Since the programmer cannot introduce new `build`s, security is assured. It is also straightforward to introduce a special typing rule into the type checker to allow `build` to be written by the programmer.

4 List Comprehensions

List comprehensions are a particularly powerful form of syntactic sugar, and have become quite widespread in functional programming languages. For example, given two lists of pairs `r1` and `r2`, each of which is intended to represent a relation, the relational join of the second field of `r1` with the first field of `r2` can be expressed like this:

```
[(x,z) | (x,y1) <- r1, (y2,z) <- r2, y1==y2]
```

This can be read as "the list of all pairs `(x,z)`, where `(x,y1)` is drawn from `r1`, `(y2,z)` is drawn from `r2`, and `y1` is equal to `y2`".

There are well established techniques for translating (or "desugaring") list comprehensions into a form which guarantees to construct only one *cons* cell for each element of the result (Augustsson [1987]; Wadler [1987]).

However these techniques view each list comprehension in isolation. Very commonly, a list comprehension is fed directly into a list-consuming function — for example, it may be appended to some other list. It is also very common for the *generators* in a comprehension to be simple list producers. For example:

```
[ f x | x <- map g xs, odd x]
```

Clearly we would like to ensure that list comprehensions are translated in a way which allows the intermediate lists between them and their producers or consumer to be eliminated. It turns out not only can we do this, but it actually makes the translation rules simpler than before, because some of the work usually done in the desugaring of list comprehensions is now done by the later `foldr/build` transformation.

Figure 3 gives the revised desugaring rules. The $\mathcal{TE}$ scheme translates expressions in a rich syntax including list comprehensions into a much simpler functional language. (Only the rules which concern us here from the $\mathcal{TE}$ scheme are given.) The first two rules deal with explicitly-specified lists, such as `[a,b,c]` and enumerations, such as `[1..4]`. The third rule deals with list comprehensions, by invoking the $\mathcal{TF}$ scheme. Notice that in each case a `build` is used, ready to cancel with the `foldr` from any list consumer.

The $\mathcal{TF}$ scheme is used only for list comprehensions, and has the following defining property:

$$\mathcal{TF}[\![\ E\ ]\!]\ c\ n = \texttt{foldr}\ c\ n\ E$$

The $\mathcal{TF}$ scheme has three cases: either the qualifiers after the "|" are empty, or they begin with a guard B, or they begin with a generator $P \leftarrow L$ (in general P can be a pattern, not just a simple variable). Notice, crucially, that in this third case, the list L is consumed by a `foldr`, so any `build` at the top of L will cancel with the `foldr`.

4.1 An example

Consider again the example given above:

```
[ f x | x <- map g xs, odd x]
```

$$\mathcal{TE} :: Expr \rightarrow CoreExpr$$

$$\begin{array}{rcl}
\mathcal{TE}[\![\,[E_1, E_2, \ldots, E_n]\,]\!] & = & \texttt{build (\textbackslash c n -> c } \mathcal{TE}[\![\,E_1\,]\!] \texttt{ (c } \mathcal{TE}[\![\,E_2\,]\!] \texttt{ } \cdots \texttt{ (c } \mathcal{TE}[\![\,E_n\,]\!] \texttt{ n)))} \\
\mathcal{TE}[\![\,[E_1 .. E_2]\,]\!] & = & \texttt{build (\textbackslash c n -> from' } \mathcal{TE}[\![\,E_1\,]\!] \texttt{ } \mathcal{TE}[\![\,E_2\,]\!] \texttt{ c n)} \\
\mathcal{TE}[\![\,[E \mid Q]\,]\!] & = & \texttt{build (\textbackslash c n ->(} \mathcal{TF}[\![\,[\,E \mid Q\,]\,]\!] \texttt{ c n))}
\end{array}$$

$$\mathcal{TF} :: Expr \rightarrow CoreExpr \rightarrow CoreExpr \rightarrow CoreExpr$$

$$\begin{array}{rcl}
\mathcal{TF}[\![\,[E \mid]\,]\!]\ c\ n & = & c\ \mathcal{TE}[\![\,E\,]\!]\ n \\
\mathcal{TF}[\![\,[E \mid B\ ,\ Q]\,]\!]\ c\ n & = & \texttt{if } \mathcal{TE}[\![\,B\,]\!] \texttt{ then } \mathcal{TF}[\![\,[E \mid Q]\,]\!]\ c\ n \texttt{ else } n \\
\mathcal{TF}[\![\,[E \mid P \leftarrow L,\ Q]\,]\!]\ c\ n & = & \texttt{foldr } f\ n\ \mathcal{TE}[\![\,L\,]\!] \\
 & & \quad \textit{where } f\ P\ b = \mathcal{TF}[\![\,[E \mid Q]\,]\!]\ c\ b \\
 & & \quad\quad f\ _\ b = b
\end{array}$$

Figure 3: List and List Comprehension Compilation Rules

The standard technology would construct an intermediate list for the result of the `map`. Using the rules in Figure 3 instead, desugaring the list comprehension will give:

```
build (\ c0 n0 ->
         foldr h n0 (map g xs)
           where
             h x b = if odd x then c0 (f x) b else b)
```

Unfolding `map` gives:

```
build (\ c0 n0 ->
         foldr h n0 (build
                (\ c1 n1 -> foldr (c1.g) n1 xs))
           where
             h x b = if odd x then c0 (f x) b else b)
```

Now we can apply the `foldr/build` rule, giving:

```
build (\ c0 n0 ->
         (\ c1 n1 -> foldr (c1.g) n1 xs) h n0
           where
             h x b = if odd x then c0 (f x) b else b)
```

Now some β-reductions can be done:

```
build (\ c0 n0 -> foldr (h.g) n0 xs
           where
             h x b = if odd x then c0 (f x) b else b)
```

Lastly, `foldr` and `build` can be unfolded, and after further simplification, we get the efficient expression:

```
h' xs
  where h' []     = []
        h' (x:xs) = if odd x'
                    then f x' : h' xs
                    else h' xs
           where x' = g x
```

5 Array Comprehensions

The Haskell language includes an array data type, in which an array is specified "all at once" in an *array comprehension*. (This contrasts with imperative languages, which usually modify arrays incrementally.) An example of an array comprehension is:

```
arr :: Array Int Int
arr = array (1,10) [ n := n * n | n <- [1..10]]
```

This defines an array `arr` with 10 elements of type `Int`, indexed with the `Int`s 1 to 10, in which each element contains the square of its index value. The part between the square brackets is just an ordinary list comprehension, which produces a list of index-value pairs (the operator `(:=)` is an infix pairing constructor). The function `array` takes the bounds of array, and the list of index-value pairs, and constructs an array from them. There is no requirement to use a list comprehension in an application of `array`; the second argument of `array` could equally well be constructed with any other list-producing function.

Clearly we would like to eliminate the intermediate list of index-value pairs altogether. After all, it is no sooner constructed by the list comprehension than it is taken apart by `array`. However, we can only use our `foldr/build` deforestation technique if `array` consumes the list of index-value pairs with `foldr`. So long as `array` is opaque we cannot eliminate the intermediate list.

So how can `array` be expressed? Presumably it must allocate a suitably sized array, and then fill in the elements of this array one by one, by working down the list of index-value pairs. In other words, *we have to build Haskell's monolithic immutable arrays on top of incrementally updateable arrays.* We have done exactly this in our compiler, as we now describe briefly.

5.1 Monads for mutable arrays

Our approach to mutable arrays is based on *monads* (Moggi [1989]; Wadler [1990]).[1] We define an (abstract) type `AT a b`, which we think of as the type of "array transformers" for arrays with indices of type `a`, and values of type `b`. That is,

[1] The approach we describe here is not the only way to implement arrays — for example it has the disadvantage of being rather sequential. The important point is that *any* implementation of array comprehensions must consume the list of index–value pairs. If (and only if) it does so with a compiler-visible `foldr`, then our transformation will eliminate the intermediate list.

a value of type `AT a b` is an array transformer which, when applied to a particular array, will transform that array in some way.

We define the following functions to manipulate array transformers:

```
createAT :: (a,a) -> AT a b -> Array a b
writeAT  :: a -> b -> AT a b

seqAT    :: AT a b -> AT a b -> AT a b
doneAT   :: AT a b
```

The function `createAT` takes the bounds of an array, and an array transformer, allocates a suitably sized array, applies the array transformer to it, and returns the resulting array. The combinator `seqAT` combines two array transformers in sequence, while `doneAT` is the identity transformer. Finally, `writeAT` is a primitive array transformer which writes a given value into the array. Array transformers can be compiled to efficient code; for example, `writeAT` can be compiled to a single assignment statement of the form

```
A[i] = v;
```

(Our compiler generates C as its target code.) For details, the interested reader is referred to Peyton Jones & Wadler [1993].

Using these four functions we define `array` as follows:

```
array bounds iv_pairs = createAT bounds action
  where
    action = foldr assign doneAT iv_pairs
    assign (a := b) n = writeAT a b `seqAT` n
```

Notice that the list of index-value pairs is now consumed by an explicit `foldr`, so that the list can be deforested if it happens to be produced by a `build`.

5.2 Deforesting an array comprehension

To illustrate the deforestation of array comprehensions, consider the example with which we began:

```
arr = array (1,10) [ x := x * x | x <- [1..10]]
```

We can now use our definition of `array` and the desugaring rules of Figure 3 to transform this to:

```
arr = createAT (1,10) action
  where
    action = foldr assign doneAT ( build
      (\ c n -> foldr h n ( build
              (\c1 n1 -> g 1
                 where
                   g x = if x > 10
                         then n1
                         else c1 x (g (x+1))))
         where
           h x b = c (x := x * x) b
      ))
    assign (a := b) n = writeAT a b `seqAT` n
```

Now we can use the `foldr/build` rule, and the usual β-reductions, to give:

```
arr = createAT (1,10) action
 where  action = g 1
        g x = if x > 10
              then doneAT
              else writeAT x (x * x) `seqAT` g (x+1)
```

This final form uses no intermediate list of pairs, and fills up the array with an efficient, tail-recursive function, `g`.

To summarise, our explicit, monadic formulation of `array` interacts very nicely with the `foldr/build` deforestation technique, so that intermediate lists in array comprehensions can be eliminated without requiring any new machinery.

6 Traversing a List Once Rather Than Twice

There are many examples of lists getting consumed twice. In his thesis, Hughes noticed that such functions can have poor space behaviour in sequential implementations. For example, the average function defined by,

```
average xs = sum xs / length xs
```

traverses the list `xs` twice. In a sequential implementation, the list `xs` cannot be consumed and discarded lazily, but at some point will exist in its entirety. Hughes solution was to propose a parallel construct to allow the computations to be interleaved (Hughes [1983]).

The method of this paper has an interesting bearing here. Re-expressing both functions in terms of `foldr` gives:

```
average xs = foldr (+) 0 xs / foldr inc 0 xs
  where inc x n = n+1
```

Now we have two `foldr` traversals of the same list. It is an easy (and generally applicable) transformation to combine these into a single traversal as follows.

```
average xs = p / q
  where (p,q) = foldr f (0,0) xs
        f x ~(p,q) = (x+p, q+1)
```

Now the list is traversed once, and may be consumed and discarded lazily. Furthermore, before the transformation the list will certainly be built, because it is used twice (the compiler would not usually substitute for `xs` if it occurs twice). After the transformation `xs` occurs only once, so its definition may be inlined, and hence may perhaps never be built at all!

Even if the consuming `foldr` cannot be paired with a `build`, the program can be more efficent. As another example consider `qsort`:

```
qsort []     = []
qsort (x:xs) = qsort [ a | a <- xs , a < x ]
                  ++ [ x ] ++
               qsort [ a | a <- xs , a >= x ]
```

Clearly xs is traversed twice. But, after translating the list comprehension into foldr, we can use the method above to obtain a single traversal.

7 Implementation and preliminary measurements

We have implemented the ideas explained above in the Glasgow Haskell compiler (Peyton Jones [1993]). The compiler passes which are affected by deforestation are as follows:

1. After typechecking, the program is desugared. This pass is now slightly simpler than before, as discussed in Section 4.
2. A sophisticated transformation pass, which we call the "simplifier", is now applied to the program. Among other things, it replaces some variables by their values (inlining) and performs β-reduction. The definitions of the standard list-processing functions in Figure 2 are made known to the simplifier so that it will inline them. The main change to the simplifier was to add the `foldr/build` transformation. Of course, at this stage, we do not inline `foldr` and `build`!
3. We now run the simplifer again, this time unfolding the definition of `build` and `foldr`. The simplifier can also perform any reductions which are thereby made possible. One important and common simplification is to reduce `foldr (:) [] xs` to `xs`.

7.1 Initial Results

To get an idea of the upper bound for the improvement which can be gained by our approach, we examine the well known 8-queens program, adapted from (Bird & Wadler [1988]). Simply printing the results would mean that I/O takes up most of the time, so instead we force evaluation by the simple expedient of adding up the result. The traditional 8-queens ran too fast on our benchmarking machine to obtain reliable timings, so we use 10-queens:

```
main = (print.sum.concat.queens) 10
  where
    queens :: Int -> [[Int]]
    queens 0 = [[]]
    queens m = [ p ++ [n] | p <- queens (m-1),
                            n <- [1..10],
                            safe p n]

    safe :: [Int] -> Int -> Bool
    safe p n = and [ (j /= n) && (i + j /= m + n)
                              && (i - j /= m - n)
                             | (i,j) <- zip [1..] p]
      where m = length p + 1
```

This example has several intermediate lists, and looks ideal for our scheme to optimise. The heart of the algorithm is checking a list of booleans (built using a list comprehension), just like the function `all` in Section 1. Because of this we expect `queens` to give a feel for an upper bound of what our optimisation can do.

The original program run without our optimisation, and averaged over several runs took 24.4 seconds and consumed 179 megabytes of heap. The transformed program under the same conditions ran about three times faster (8.8 seconds) and allocated only 20% as much heap (36 megabytes). Similar measurements for more realistic programs is still ongoing, but our preliminary results suggest (unsurprisingly) that the speedups are much more modest.

8 Further Work

There are several directions in which this work can be developed.

8.1 Extending the scope of the transformation

At present we can only use `build` in the definitions of "built-in" standard list-processing functions. There are two ways in which this may be improved. The first is to make `build` available to programmers. This would necessitate extending the type checker to ensure that its calls are valid, but that poses no particular difficulty.

More challenging is for the compiler to spot functions which may be defined in terms of `build` and `foldr` and redefine them accordingly. This is likely to be quite feasible, so long as the compiler is only expected to notice the more obvious examples. This has the benefit that the programmer need know nothing about optimisations internal to the compiler, and may write programs in whatever style is most appropriate for the application.

Another current limitation is that we do not attempt to perform deforestation across function boundaries. That is, if a function `f` has a list as its result, and the transformation system does not inline the function at its call site(s), then no deforestation will occur between `f` and its callers. Indeed, it seems impossible to do so without changing the type of the result of `f`. Nonetheless, it is interesting to speculate whether some systematic transformation to `build`-like form of such list-returning functions would be possible.

Experiments with inter-function optimisations have been done by hand with encouraging performance results. For example 10-queens showed a further significant speedup when inter-function optimisations were used in conjunction with the `foldr/build` rule. However the effects of transforming *all* list producing functions in this way may be detrimental to the overall performance, because not all lists are consumed using a `foldr` based function. Some sort of analysis technique could help here, but some code duplication looks inevitable.

Another obvious extension is to generalise the idea to arbitrary algebraic data types. Both `foldr` and `build` generalise very naturally to these other types, and corresponding transformations apply. The categorical properties of these operators (the so-called *catamorphisms*) have been well studied. For example, Fokkinga et al. [1991] show that any attribute grammar may be expressed by a single catamorphism.

8.2 Dynamic deforestation

The transformation we describe works only for *statically-visible* compositions of `foldr` and `build`. Would it be possible also to spot such compositions *at run time*? What would be required would be an extra constructor in the list data type, like this:

```
data List a
        = Nil
        | Cons a (List a)
        | Build ((a -> b -> b) -> b -> b)
```

The new `Build` constructor has exactly the type of `build` given earlier, namely:

$$\texttt{Build} : \forall\alpha.(\forall\beta.(\alpha \to \beta \to \beta) \to \beta \to \beta) \to \mathit{List}\ \alpha$$

Now, the function `foldr` can be defined as follows:

```
foldr k z Nil         = z
foldr k z (Cons x xs) = k x (foldr k z xs)
foldr k z (Build g)   = g k z
```

If `foldr k z` is applied to a list which happens to have been built with `Build`, then it applies the function inside the `Build` to `k` and `z`. If any function which expects to a list to be either a `Cons` or `Nil` finds a `Build` instead, it just applies the function inside the `Build` to `Cons` and `Nil`, and the result now really will be a `Cons`/`Nil` list.

Among other things, this turns out to implement "bone idle append", which is part of the folk lore (Sleep & Holmstrom [1982]). It is well known that left-bracketed associations of append, such as `(xs++ys)++zs`, are very inefficient to execute, because `xs` gets traversed twice. It has often been suggested that `++` should at runtime take a look at its first argument to see if it looks like another application of `++`, and if so simply rearrange itself to the more efficient `xs++(ys++zs)`.

A closely related trick is the "top-level append optimisation". When outputting a list of characters to a file, if the output mechanism recognises that it is being asked to output `(xs++ys)`, then it does not actually need to append `xs` and `ys`; rather it can simply output them one after the other. This, too, is subsumed by our dynamic deforestation technique, *provided that the list is consumed by a* `foldr`, which is the case for our monadic form of I/O[2] (Peyton Jones & Wadler [1993]).

9 Limitations

A crop of limitations arises when we consider functions such as `tail` and `foldr1`. Neither of these functions treat all the *cons* cells in their inputs identically. In particular, `tail` treats the first *cons* specially, and `foldr1` the last. To try to adapt the `foldr/build` transformation to cases such as these seems to add so much complication that the original simplicity is lost. It seems reasonable that functions which do not have regular recursive patterns really need full scale deforestation using fold/unfold transformations, and that we should not expect to find short cuts in these cases. Of course, from an engineering point of view, such traversal may be sufficiently infrequent to mean that it is simply not worth while going to any effort to remove such intermediate lists.

A more serious limitation involves `zip`. Although considering `zip` as a list producer is straightward, there seems to be no easy way to extend the technique here so that *both* input lists to `zip` may be deforested. Note that it is easy to ensure that one or other of the input lists is available for deforestation as `zip` may be defined in terms of `foldr` as follows.

```
zip xs ys = foldr f (\ _ -> []) xs ys
  where f x g   []     = []
        f x g (y:ys) = (x,y) : g ys
```

The `foldr` traverses `xs` constructing a function which takes `ys` and builds the zipped list. The list `xs` may disappear using the `foldr/build` transformation, but `ys` never will.

10 Related Work

This concept of parameterising over a list has been considered before. Parameterising over the *nil* of a list has been proposed as a possible optimisation over traditional lists (Hughes [1984]). In particular, this leads to constant-time append operations.

This idea of optimising append using parameterisation over *nil* was taken one step further by Wadler [1987], who described a global transformation which removes many appends from a program using this improved representation of lists.

After completing this paper we were introduced to an astonishing paper by Burge [1977]. Despite the early date of this paper, Burge presents many of the essential ideas we have described, including the key step of parameterising over *cons* and *nil*. However, he requires individual rules for each list producing expression, therefore stopping just short of giving the `foldr/build` rule, and of course his work predates list and array comprehensions.

It is interesting to note that the compositional style of programming is not restricted to lazy functional languages. Waters introduces a data type called *series* into imperative languages (Pascal and LISP), which behave exactly like lazy lists except *that they are removed by compile time transformation*, so never appear in the final object code (Waters [1991]). A number of fairly stringent conditions are imposed which guarantee the complete compile-time removal of series. Were we to apply the same restrictions to lists, we too could guarantee a "list-less" final program using the techniques here.

Anderson & Hudak [1989] discuss the compilation of Haskell array comprehensions, but their main focus is on the data-dependence analysis required to compile *recursive* array definitions into *strict* computations. The important thing for their work is that the computation of an array element must depend only on elements which have already been computed. This question is quite orthogonal to our work.

Acknowledgements

The implementation of the Glasgow Haskell compiler is a team project. Apart from ourselves, the main participants have been Will Partain, Cordy Hall, Patrick Sansom and André Santos. In particular, the simplifier we modified was written by André, we would like to thank him for directing us around it. Thanks, also, to Phil Wadler for help with the "free theorem" for `build`.

This work was done as part of the SERC GRASP project.

[2] More commonly, the mechanism which consumes the list of characters to be printed is part of the runtime system.

It was also supported by the SERC studentship, number 91308622.

Bibliography

S Anderson & P Hudak [March 1989], "Efficient compilation of Haskell array comprehensions," Dept Comp Sci, Yale University.

L Augustsson [1987], "Compiling lazy functional languages, part II," PhD thesis, Dept Comp Sci, Chalmers University, Sweden.

R Bird & PL Wadler [1988], *Introduction to Functional Programming*, Prentice Hall.

WH Burge [Oct 1977], "Examples of program optimization," RC 6351, IBM Thomas J Watson Research Centre.

RM Burstall & John Darlington [Jan 1977], "A transformation system for developing recursive programs," *JACM* 24, 44–67.

L Cardelli & G Longo [Oct 1991], "A semantic basis for Quest," *Journal of Functional Programming* 1, 417–458.

WN Chin [March 1990], "Automatic methods for program transformation," PhD thesis, Imperial College, London.

K Davis [Sept 1987], "Deforestation: Transformation of functional programs to eliminate intermed-ate trees," MSc Thesis, Programming Research Group, Oxford University.

J Fairbairn [May 1985], "Design and implementation of a simple typed language based on the lambda calculus," TR 75, Computer Lab, Cambridge.

AB Ferguson & PL Wadler [1988], "When will deforesation stop?," in *Functional Programming, Glasgow 1988*.

MM Fokkinga, E Meijer, J Jeuring, L Meertens [1992], "FRATS: a parallel reduction strategy for shared memory," *The Squiggolist* 2, 20–26, KG Langendoen & WG Vree.

P Hudak, SL Peyton Jones, PL Wadler, Arvind, B Boutel, J Fairbairn, J Fasel, M Guzman, K Hammond, J Hughes, T Johnsson, R Kieburtz, RS Nikhil, W Partain & J Peterson [May 1992], "Report on the functional programming language Haskell, Version 1.2," *SIGPLAN Notices* 27.

RJM Hughes [July 1983], "The design and implementation of programming languages," PhD thesis, Programming Research Group, Oxford.

RJM Hughes [Oct 1984], "A novel representation of lists and its application to the function 'Reverse'," PMG-38, Programming Methodology Group, Chalmers Inst, Sweden.

S Marlow & PL Wadler [1993], "Deforestation for higher-order functions," in *Functional Programming, Glasgow 1992*, J Launchbury, ed., Workshops in Computing, Springer Verlag.

R Milner [Dec 1978], "A theory of type polymorphism in programming," *JCSS* 13.

E Moggi [June 1989], "Computational lambda calculus and monads," in *Logic in Computer Science, California*, IEEE.

SL Peyton Jones [1993], "The Glasgow Haskell compiler: a technical overview," in *Joint Framework for Information Technology Technical Conference, Keele*.

SL Peyton Jones & PL Wadler [Jan 1993], "Imperative functional programming," in *20th ACM Symposium on Principles of Programming Languages, Charleston*, ACM.

JC Reynolds [1983], "Types, abstraction and parametric polymorphism," in *Information Processing 83*, REA Mason, ed., North-Holland, 513–523.

MR Sleep & S Holmstrom [May 1982], "A short note concerning lazy reduction rules of APPEND," University of East Anglia.

VF Turchin [1988], "The algorithm of generalization in the supercompiler," in *Partial Evaluation and Mixed Computation*, Bjørner, Ershov & Jones, eds., North-Holland.

PL Wadler [1987], "List Comprehensions," in *The Implementation of Functional Programming Languages*, SL Peyton Jones, ed., Prentice Hall, 127–138.

PL Wadler [1989], "Theorems for free!," in *Fourth International Conference on Functional Programming and Computer Architecture, London*, MacQueen, ed., Addison Wesley.

PL Wadler [1990], "Deforestation: transforming programs to eliminate trees," *Theoretical Computer Science* 73, 231–248.

PL Wadler [Dec 1987], "The concatenate vanishes," Dept of Computer Science, Glasgow University.

PL Wadler [June 1990], "Comprehending monads," in *Proc ACM Conference on Lisp and Functional Programming, Nice*, ACM.

R Waters [Jan 1991], "Automatic Transformation of Series Expressions into Loops," *ACM TOPLAS* 13, 52–98.