Shortcut Fusion for Accumulating Parameters & Zip-like Functions

Josef Svenningsson
Chalmers University of Technology
josefs@cs.chalmers.se

Abstract

We present an alternative approach to shortcut fusion based on the function `unfoldr`. Despite its simplicity the technique can remove intermediate lists in examples which are known to be difficult. We show that it can remove all lists from definitions involving `zip`-like functions and functions using accumulating parameters.

Categories and Subject Descriptors

D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.3.4 [**Programming Languages**]: Optimization; F.3.3 [**Studies of Program Constructs**]: Program and recursion schemes; I.1.4 [**Symbolic and Algebraic Manipulation**]: Applications

General Terms

Languages, Algorithms

Keywords

Deforestation, functional programming, intermediate data structures, program transformation

1 Introduction

Functional programmers like to write programs by composing small, highly parameterised functions. When composed, these functions use intermediate data structures to communicate with each other. These intermediate data structures are often lists. A standard example of such a function is the following:

```
sumTo n = sum (map square [1..n])
```

The example is written in Haskell [JH99b]. We will use Haskell throughout this paper. This function produces two intermediate lists, one from the expression `[1..n]` which is immediately consumed by `map`. Next, `map` produces a list which is immediately consumed by `sum`. Both these lists are intermediate structures which do not form part of the result.

It is by now well known that many intermediate lists can be removed automatically by using program transformations and a considerable amount of research has gone into developing more powerful and easily applicable techniques, e.g. [Wad90, Mar96, TM95, CDPR99, Ham01, Chi99].

Perhaps the most successful method for removing intermediate data structures is shortcut fusion [GLJ93]. It relies on writing list processing functions using two special functions `foldr` and `build`. Whenever a list is produced by `build` and consumed by `foldr`, that list can be removed. The technique has been shown to work on many practical examples and it is very easy to implement. It is incorporated in the Glasgow Haskell Compiler [PJTH01].

Despite its success, shortcut fusion (henceforth called the foldr/build rule) has its shortcomings. Firstly, it cannot remove both lists of the function `zip`. Secondly it cannot handle functions which consume their lists using accumulating parameters.

In this paper we present the destroy/unfoldr rule as a new means for shortcut fusion. It has the following characteristics:

- The technique can remove all argument lists from a function which consumes more than one list, for example the function `zip`. In the expression `zip [1..n] [1..n]` our method will be able to remove all intermediate lists. The problem with `zip`-like functions has been one of the main criticisms against the foldr/build rule.
- The technique can remove intermediate lists from functions which consume their lists using accumulating parameters. For example, all intermediate lists can be removed from the function `sumTo`, *even when* `sum` *is defined using an accumulating parameter*. Accumulating parameters are known to be problematic when fusing functions and most standard techniques suffer from the inability to fuse functions defined using them.
- Like the foldr/build rule, our method is simple. It can be implemented in the same manner and is therefore a good candidate for incorporating in a compiler.

It should be noted that the transformation itself is not new. It was noted by Takano and Meijer that the foldr/build rule has a dual [TM95]. They did not, however, consider this transformation but instead focused on using hylomorphisms to express fusion.

ICFP'02, October 4-6, 2002, Pittsburgh, Pennsylvania, USA.

In this paper we will only consider lists when we want to remove intermediate data structures. But most things we present generalises (like the foldr/build rule) directly to other data types.

The paper is organised as follows. We will begin by recapitulate the foldr/build rule in section 2. Section 3 explains the destroy/unfoldr rule and show how list processing functions can be defined in terms of the functions `destroy` and `unfoldr`. In the next two sections we will show how the destroy/unfoldr rule can remove intermediate lists from definitions involving `zip`-like functions (Section 4) and accumulating parameters (Section 5). Section 6 discusses related work and Section 7 concludes. Some correctness issues of shortcut fusion will be discussed in an appendix.

2 foldr/build

foldr/build fusion is perhaps the most successful technique for removing intermediate data structures. But how does it work?

The whole story begins with two functions, `foldr` and `build`. Let's start by looking at `foldr`.

The function `foldr` is rather well-known to the functional programming community. It has several names such as `reduce` and `accumulate`. It is known to be the catamorphism for lists [MFP91]. It can be defined as follows:

```
foldr f n [] = n
foldr f n (x:xs) = f x (foldr f n xs)
```

Informally `foldr` goes through the list replaces every cons by its first argument (`f`) and replaces nil by its second argument (`n`).

Next, we look at the function `build` which is less well-known and rather specific to shortcut fusion. Here is how it is defined:

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

The important thing with `build` is not the function itself but its argument `g`. It is a function which is supposed to produce a list. But it may not do so using the list constructors (`:`) and `[]` (cons and nil) directly. Instead it must use whatever values are passed to it as arguments to construct the list. This is because after applying the foldr/build rule, `g` might no longer be producing lists at all. We will shortly come back to how we can ensure that `g` doesn't use the list constructors directly. We note that `build` applied to `foldr` gives the identity function, i.e.:

```
build foldr == id
```

The next part is the actual rule which can remove intermediate data structures, the foldr/build rule. The beauty is in its simplicity:

```
foldr c n (build g) ==> g c n
```

This rule, as it stands, is, however, not correct. The correctness of the rule relies on the fact that the function `g` does not use the list constructors internally while producing the list. Instead it *must* use its arguments to construct the list. The idea is then that if we pass it something different than the list constructors, then another kind of value is produced. And this is exactly what the foldr/build rule does. So how can we make sure that `g` does not use the list constructors to construct its result list? It can be ensured by restricting `g`'s type. But in actual implementations it turns out to be more convenient to give `build` a more refined type than the inferred one:

```
build :: (forall b. (a -> b -> b) -> b -> b) -> [a]
```

(Haskell does not have types of higher rank but several implementations provide support for this.) From this type we can see that `g` must be polymorphic in its result type and the only way that it can produce it is to use its arguments. Using this type we can ensure that `g` is sufficiently polymorphic.

Initially, work on model theory for polymorphic lambda calculus was used to argue for the correctness of the rule [Wad89]. Recently, a proof based on operational techniques has been given [Joh01].

In order for the rule to apply the programmer needs to define his/her functions using `foldr` and `build`. Only then will the rule apply. This has turned out not to be a big burden since many functions in the standard libraries can be written in this style and therefore it suffices to use those functions to be certain that intermediate lists are removed.

It should be noted that the foldr/build rule is in itself rather harmless. It relies heavily on other transformations which enable the rule to apply. The most important are inlining of functions and β-reduction. In some cases it also relies on arity analysis [Gil96].

In this paper we will explicitly write out in each step what other transformations are used to enable fusion in order to show that there is no magic going on. Most of the transformations that we will use are completely standard and are implemented in the Glasgow Haskell Compiler [PJS98]. If this is not the case we will briefly explain them.

2.1 Good producers and consumers

When talking about short cut fusion it is often handy to classify functions as *good producers* and/or *good consumers*. This terminology was introduced in [Gil96]. We will here define what they mean and get some intuition about them.

The idea behind good producers and consumers is the following: Whenever a good consumer is applied to a good producer the intermediate data structure between the two can be removed. This can be achieved by defining good producers in terms of `build` and good consumers in terms of `foldr`.

More formally a *good producer* is a function which for some expression `e` and arguments `a1` to `an` is defined as follows:

```
f a1 .. an = build e
```

In other words a good producer must be defined directly in terms of `build`.

Good consumers are a little trickier to define. A function may take several lists as arguments but need not be a good consumer in all of these arguments. It is therefore necessary to consider a function to be a good consumer in certain arguments. This gives rise to the following definition. Consider the following function definition:

```
f a1 .. ak .. an = e
```

we will say that `f` is a *good consumer in ak* if `ak` occurs once in `e` as the third argument to `foldr`.

It is actually possible to loosen the requirements on a good consumer but the current definition will suffice in this paper.

Now we can see that if we have an application of a good consumer

applied to a good producer and we inline both function definitions we can directly apply the foldr/build rule and remove the intermediate list. We will see several examples of this in the coming sections.

The informed reader notes that these definitions differs somewhat from the definitions in [Gil96]. For the purpose of this paper this difference is unimportant.

When we move to the next section we will revise the notion of good consumers and producers to fit our purposes.

2.2 Limitations

It is well known that the foldr/build rule is unable to handle zip-like functions and Gill describes it as a "significant shortcoming" in his thesis [Gil96]. The reason is that `foldr` only traverses one list at a time. However, it is quite possible to remove *one* of the two lists which are fed to `zip`.

Functions that consume lists using accumulating parameters also cause problems for the foldr/build rule. The reason is again the function `foldr`. All good consumers have to be written in terms of `foldr` and `foldr` simply doesn't handle accumulating parameters. Some readers might object to this because the function `foldl` is defined using `foldr` and it uses accumulating parameters. Using this definition buys us nothing however. It introduces suspended function calls isomorphic to the removed list and so we have gained nothing.

The foldr/build rule has other shortcomings as well but the ones we just mentioned are the ones we will tackle in this paper.

3 destroy/unfoldr

The optimisation that we present in this paper has three important ingredients that we will explain in this section. But before we plunge into our explanation we would like to emphasise the fact that the material presented in this section is not new. But, although not new, many of the things we present here is not known to a wider audience and they are vital to understand the contribution of this paper.

3.1 unfoldr

Our first key ingredient is the function `unfoldr`. This is a rarely used function which can be found in the Haskell Library report [JH99a]. As the name reveals it does the opposite of `foldr`, it constructs lists instead of consuming them.

```
unfoldr :: (b -> Maybe (a,b)) -> b -> [a]
unfoldr f b = case f b of
                Nothing    -> []
                Just (a,b') -> a : unfoldr f b'
```

An operational intuition of a call `unfoldr f b` is as follows: `unfoldr` is given an initial state `b`. The function `f` is applied to `b` to determine whether we shall produce more of the output list. If `f b` returns `Nothing` then the end of the list is returned. On the other hand if the list should be longer, `f b` returns the value `Just (a,b')` where `a` is the new element of the list and `b'` is the next state we shall use to produce the rest of the list.

In some examples we will inline `unfoldr`. But inlining recursive functions can be problematic. When we want to use `unfoldr` for this purpose we will use the following definition which is non-recursive but uses a locally defined recursive function which does the job:

```
unfoldr f b = go b
  where go b = case f b of
                 Nothing -> []
                 Just (a,b') -> a : go b'
```

This version of `unfoldr` can easily be inlined.

3.2 destroy

Our next key ingredient is also a function. We have chosen to baptise it *destroy* since it consumes lists [1]. It does so in a fashion tailored to fit together with `unfoldr`. Its definition is as follows:

```
destroy g xs = g listpsi xs
  where listpsi :: [a] -> Maybe (a,[a])
        listpsi [] = Nothing
        listpsi (x:xs) = Just (x,xs)
```

The reader might wonder what the type of `destroy` is. We will have reason to come back to this in the next subsection.

Just as `build` applied to `foldr` yields the identity function we have a similar law for `destroy` and `unfoldr`. We have that `destroy` *composed* with `unfoldr` yields the identity. Stated algebraically this looks like:

```
destroy . unfoldr == id
```

This follows immediately from the fact that `unfoldr listpsi == id`.

The interesting thing with `destroy` is, as with `build`, not the function itself but its argument. `g` is a function which consumes a list, but it must not do so by pattern matching on the list. Instead it is passed a function through which it can inspect the list, thus we have created a kind of "view"[Wad87] of lists for `g`.

3.3 The destroy/unfoldr rule

The third ingredient is a transformation rule using both `destroy` and `unfoldr`. We call it the destroy/unfoldr rule and it looks like this:

```
destroy g (unfoldr psi e) ==> g psi e
```

As with the foldr/build rule, this rule does not hold unconditionally but we must require that the type of `g` is `forall a.(a -> Maybe (b,a)) -> a -> c)`[2] for some types `b` and `c`. This can be ensured by giving `destroy` the following type:

```
destroy :: (forall a. (a -> Maybe (b,a)) -> a -> c)
           -> [b] -> c
```

The idea with the destroy/unfoldr rule is (as with the foldr/build rule) to define functions in terms of `destroy` and `unfoldr`. We

[1] In his thesis Gill calls this function `unbuild` [Gil96]. This was unknown to us at the time we (re-)discovered this function and we have chosen to stick to the name we came up with.

[2] Some readers might be worried about the use of the `Maybe` type. Are we removing lists just to introduce another structure? No, the `Maybe` type is only transient and will be removed when the transformation machinery is finished.

can then inline these functions and apply the destroy/unfoldr rule to remove intermediate lists. Next, we look into the problem of defining list processing functions using `destroy` and `unfoldr`.

We believe that the destroy/unfoldr rule can be proved using similar techniques as in [Joh01]. This is, however, a substantial exercise and we leave it as future work.

3.4 Expressing list functions using unfoldr and destroy

If we want to use the destroy/unfoldr rule to remove intermediate lists we must express our functions in terms of `unfoldr` and `destroy`. Intuitively this should be quite easy but it will soon become evident that there is more to it than meets the eye.

First we will revise the notion of good producers and consumers. A *good producer* is a function which for some expressions `psi`, `e` and arguments `a1` to `an` is defined as follows:

```
f a1 .. an = unfoldr psi e
```

In order to get a feeling for how the destroy/unfoldr rule works let us first look at some simple examples. The function `enumFromTo` takes two integers `n` and `m` and returns a list of all integers between and including `n` and `m`. Since it is a list producing function we should define it using `unfoldr`. This can be done as follows:

```
enumFromTo n m =
  unfoldr (\i -> if i > m
                 then Nothing
                 else Just (i,succ i)) n
```

An interesting example (at least in this context) of a function which consumes a list is `foldr`. It can be defined using `destroy` in the following way:

```
foldr k z xs = destroy foldrDU xs
  where foldrDU psi xs =
          case psi xs of
            Nothing -> z
            Just (x,ys) -> k x (foldrDU psi ys)
```

It should be noted that we can define a large class of list consumers in terms of `destroy`. We will have more to say about this in section 5.

Next, we turn to the task of defining functions which both produce and consume lists, e.g. `map`. The function `map` can be defined using `foldr` and, as we saw, `foldr` could easily be written as a good consumer. We shall therefore begin with a definition of `map` where it is only a good consumer and then refine it:

```
map f xs = destroy mapDU ls
  where mapDU psi xs =
          case psi xs of
            Nothing -> []
            Just (x,ys) -> f x : mapDU psi ys
```

When we want to define a list producing function using `unfoldr` we should aim at replacing `[]` with `Nothing` and `:` with `Just`. This is not always possible because the second argument to `:` needs to be a recursive call. In this case, however, it is perfectly possible and we end up with the following definition:

```
map f xs = destroy (\psi a ->
                    unfoldr (mapDU psi) a) xs
  where mapDU psi xs =
          case psi xs of
            Nothing -> Nothing
            Just (x,ys) -> Just (f x,ys)
```

There is a problem with this definition however. It is not a good producer since `unfoldr` is not at the outermost level of the function definition. There is a way to write `map` so that it becomes a good producer but then it is not a good consumer. There doesn't seem to be a way we can define `map` so that it is both a good producer and a good consumer.

There is, however, a way out of our problems. Consider what happens when we want to fuse two `map` functions that sit next to each other like this:

```
map f (map g xs)
```

When we inline both occurrences of `map` and α-rename we get the following expression:

```
destroy (\psi a ->
           unfoldr (mapDUf psi) a)
         (destroy (\psi a ->
                     unfoldr (mapDUg psi) a) xs)
  where mapDUf = ...
        mapDUg = ...
```

We can see that the inner `destroy` prevents the `unfoldr` from contacting the outer `destroy` and allowing the destroy/unfoldr rule to apply. Our solution to this is another rule which lets a `destroy` move inside another `destroy` and hopefully encounter an `unfoldr`. The rule looks like this:

```
destroy g (destroy g' ls) ==>
  destroy (\psi a -> destroy g (g' psi a)) ls
```

We will call this rule the destroy/destroy rule. It can easily be shown correct by unfolding the definition of `destroy`. The idea with the rule is that if `g'` happens to be a function defined using `unfoldr` then we will bring it together with the outer `destroy` using the above rule.

Now we can continue with our example involving the two `map`s. Here is what we will get if we apply our new destroy/destroy rule:

```
destroy (\psi1 a1 ->
           destroy (\psi a ->
                      unfoldr (mapDUf psi) a)
                   ((\psi a ->
                       unfoldr (mapDUg psi) a)
                    psi1 a1)) xs
  where mapDUf = ...
        mapDUg = ...
```

Performing two β-reductions will give us a possibility to apply the destroy/unfoldr rule:

```
destroy (\psi1 a1 ->
           destroy (\psi a -> unfoldr (mapDUf psi) a)
                   (unfoldr (mapDUg psi1) a1)) xs
  where mapDUf = ...
        mapDUg = ...
```

Applying the destroy/unfoldr rule and performing two β-reductions gives us:

```
destroy (\psi1 a1 ->
          unfoldr (mapDUf (mapDUg psi1)) a1) xs
  where mapDUf = ...
        mapDUg = ...
```

Inlining and simplifying `mapDUf` and `mapDUg` will give a `map` function which applies the function `f.g` to each element without creating any intermediate data structure.

We have seen how we can define `map` in terms of `destroy` and `unfoldr`. Some other common list processing functions can be found in figure 1.

4 zip fusion

One of the criticisms that have been raised against foldr/build fusion is the following: Suppose we have a function `f` which recurses over more than one list simultaniously. We will henceforth call such functions *zip-like*. In the foldr/build framework we can make `f` a good producer in *only one of* its arguments[3]. In this section we show how destroy/unfoldr fusion solves this problem gracefully. We do this by means of an example. Although we only give an example it should be noted that the result is completely general.

The first thing we have to do is define the function `zip` in terms of `destroy` and `unfoldr` keeping in mind that we would like to fuse both its input lists. This turns out to be quite easy.

```
zip xs ys =
 destroy (\ psi1 e1 ->
          destroy (\ psi2 e2 ->
                   unfoldr (zipDU psi1 psi2) (e1,e2)
                  ) ys
         ) xs
  where zipDU psi1 psi2 (e1,e2) =
              case psi1 e1 of
               Nothing -> Nothing
               Just (x,xs) ->
                case psi2 e2 of
                 Nothing -> Nothing
                 Just (y,ys) -> Just ((x,y),(xs,ys))
```

We will now see how this definition of `zip` enables us to remove the intermediate lists of both arguments. Consider the following function:

```
ascii_table = zip (enumFromTo 'A' 'Z')
                  (enumFromTo 65 90)
```

Inlining `zip` and `enumFromTo` gives us:

```
ascii_table =
  destroy (\psi1 e1 ->
           destroy (\psi2 e2 ->
                    unfoldr (zipDU psi1 psi2)
                            (e1,e2))
                   (unfoldr (\i ->
                          if i > 90
                          then Nothing
                          else Just (i,succ i)) 65))
          (unfoldr (\i -> if i > 'Z'
                          then Nothing
                          else Just (i,succ i)) 'A')
  where zipDU = ...
```

This gives us two opportunities to apply the destroy/unfoldr rule. Doing so will give:

```
ascii_table =
  unfoldr (zipDU (\i -> if i > 'Z'
                        then Nothing
                        else Just (i,succ i))
                 (\i -> if i > 90
                        then Nothing
                        else Just (i,succ i)))
          ('A',65)
  where zipDU = ...
```

Now, let us inline `zipDU`. After that, performing two β-reductions, case-of-if and case-of-known will give us:

```
ascii_table =
  unfoldr (\(e1,e2) ->
            if e1 > 'Z'
            then Nothing
            else
              if e2 > 90
              then Nothing
              else Just ((e1,e2),(succ e1,succ e2)))
          ('A',65)
```

Inlining `unfoldr` (and translating the lambda-pattern to a case) will yield:

```
ascii_table = go ('A',65)
  where go b =
          case (\a ->
             case a of
               (e1,e2) ->
                  if e1 > 'Z'
                  then Nothing
                  else
                    if e2 > 90
                    then Nothing
                    else Just ((e1,e2)
                              ,(succ e1
                               ,succ e2))) b of
            Nothing -> []
            Just (a,b') -> a : go b'
```

This in turn can be simplified (via a β-reduction and performing case-of-case and case-of-known transformations) to:

```
ascii_table = go ('A',65)
  where go b =
    case b of
      (e1,e2) ->
         if e1 > 'Z'
         then []
         else if e2 > 90
              then []
              else (e1,e2) : go (succ e1,succ e2)
```

As we can see both intermediate lists have been removed.

[3] It should be noted that even though `f` can only be a good consumer in one of its arguments this argument need not be fixed. The important thing is that `f` can only be a good consumer for one argument at a time.

```
map f xs                = destroy (\psi a -> unfoldr (mapDU psi) a) xs
  where mapDU psi xs    = case psi xs of
                            Nothing -> Nothing
                            Just (x,ys) -> Just (f x,ys)
filter p xs             = destroy (\psi a -> unfoldr (filterDU psi) a) xs
  where filterDU psi xs = case psi xs of
                            Nothing -> Nothing
                            Just (b,ys) -> if p b
                                           then Just (b,ys)
                                           else filterDU psi ys
foldr f z xs            = destroy foldrDU xs
  where foldrDU psi xs  = case psi xs of
                            Nothing -> z
                            Just (x,ys) -> f x (foldrDU psi ys)
enumFromTo n m          = unfoldr (\i -> if i > m then Nothing else Just (i,succ i)) n
repeat x                = unfoldr (\a -> Just (x,a)) undefined
[]                      = unfoldr (const Nothing) undefined
```

Figure 1. Some standard list processing functions defined using `destroy` and `unfoldr`

5 Fusion with accumulating parameters

Some functions that consume lists have to be defined using an accumulating parameter. Others are defined using an accumulating parameter for efficiency reasons. If we want to fuse such functions to remove intermediate lists we currently have to use rather sophisticated methods [CDPR99, VK01]. In this section we show how destroy/unfoldr fusion can achieve this elegantly in some cases. Again we note that even though we only give an example the result is generally applicable.

Many functions consuming lists using accumulating parameters can be expressed in terms of two higher order functions `foldl` and `foldl'`. These functions are well known to the functional programmer. The latter function is a strict version of the first which allows the compiler to generate code that is usually more efficient. We will use `foldl'` as an example to show how the destroy/unfoldr rule can deal with accumulating parameters.

For our purposes we will define `foldl'` in a rather roundabout way, having two locally defined functions `foldlDU` and `foldlDU'`. Using `foldl'` on this form simplifies the transformations that we want to do.

```
foldl' f b xs = destroy (foldlDU b) xs
  where foldlDU acc psi xs = foldlDU' acc xs
          where foldlDU' acc xs =
                  case psi xs of
                  Nothing -> acc
                  Just (a,ys) ->
                    let acc' = f acc a
                    in seq acc' (foldlDU' acc' ys)
```

`seq` is a function which evaluates its first argument and returns its second argument after evaluating it. In this way `acc'` will be evaluated before `foldlDU'` is called recursively.

Now, to our example. Consider the following function definition:

```
bar f b n m = foldl' f b (enumFromTo n m)
```

The function `enumFromTo` produces a list which is consumed by `foldl'`. The goal as in all examples is to remove this intermediate list.

We begin by inlining `foldl'` and `enumFromTo`. This will yield:

```
bar f b n m =
  destroy (foldlDU b)
          (unfoldr (\i ->
                      if i > m
                      then Nothing
                      else Just (i,succ i)) n)
  where foldlDU acc psi xs = foldlDU' acc xs
          where foldlDU' acc xs = ...
```

This gives us an opportunity to apply the unfoldr/destroy rule. Doing so will give us:

```
bar f b n m = foldlDU b (\i ->
                           if i > m
                           then Nothing
                           else Just (i,succ i)) n
  where foldlDU acc psi xs = foldlDU' acc xs
          where foldlDU' acc xs = ...
```

The next thing we will do is to inline `foldlDU`. After applying the transformations case-of-case, case-of-known and β-reduction we will end up with the following definition:

```
bar f b n m = foldlDU' b n
  where foldlDU' acc xs =
          if xs > m
          then acc
          else let acc' = f acc xs
               in seq acc' (foldlDU' acc' (succ xs))
```

Assuming that the compiler can spot that `foldlDU'` is strict in its second argument this version of `bar` is as efficient as we may hope for. Most notably, all intermediate structures between `foldl'` and `enumFromTo` have been removed.

6 Related work

Removing intermediate data structures is a popular research subject and has received quite a lot of attention e.g. [Wad84, Wad90, Chi92, GLJ93, SF93, TM95, Feg96, Chi99].

Fegaras, Sheard and Zhou were (to our knowledge) the first group to attack to problem of fusing functions which recurse over multiple data [FSZ94]. They extended their previous method for fusing

functions [SF93] to handle this wider class of functions. They rely on a normalisation algorithm which transforms function definitions. Their approach is rather powerful but it is unclear how well it would work in a compiler with real programs as input.

Later, Takano and Meijer responded to the original shortcut fusion paper by generalising the fusion law using hylomorphisms [TM95]. They start by observing that the foldr/build rule has a dual, the destroy/unfoldr rule we use in this paper. However, they do not study it any further but instead focus on fusing functions expressed as hylomorphisms. It should be noted that although hylomorphisms are generalisations of both `foldr`- and `unfoldr`-like functions their transformation does not generalise the corresponding transformations. The problem is that hylomorphisms are unable to express functions such as those using accumulating parameters, which can be expressed with `destroy`. In their paper they show how their method can fuse all lists arguments to the function `zip`. This claim has, however, been criticised in the paper [HIT96] which develops more theory to be able to fuse `zip`-like functions.

Recently there has been work on trying to extend the foldr/build rule to handle `zip`-like functions [LKS00]. This approach has modified the `foldr` function to use a notion of hyperfunctions. This, however, makes the initial simplicity of the foldr/build rule disappear since the original `foldr` function is not usable any more and it also requires all list processing functions to have their types changed.

Accumulating parameters are known to be problematic when removing intermediate data structures and most techniques fail for such functions. Ideas which have been developed to tackle this weakness are to fuse attribute grammars [CDPR99] or macro tree transducers [VK01]. These methods can handle quite a large class of functions and deal easily with functions using accumulating parameters. However, these approaches are rather heavyweight. The former method require that functions are rewritten into attribute grammars which are quite different from the functional language being transformed. The transformation is then applied on these intermediate forms and then translated back. In the latter method the tranformation work on a restricted form of functions which have to be identified before transforming them. In both cases the transformations involved are non-trivial. This is in contrast to our method which is extremely simple, although it can probably not handle as large a class of functions.

Recently Voigtländer has proposed a methodology for removing intermediate data structures from programs by abstracting over these operations [Voi02]. The key idea is to use a generalised form of `build` which abstracts not only the list constructors but other list manipulating functions as well. Using this technique makes it possible to remove intermediate structures in many cases. Although quite simple and elegant the technique does require the programmer to change the functions in which fusion is to take place. The author suggests that this may be automated in a similar fasion as [Chi99]. We believe that it is possible to dualise his result in the same way as we have dualised shortcut fusion in this paper.

This paper also gives fuel to the opinion that the function `unfoldr` is greatly under-appreciated [GJ98]. Gibbons and Jones note that `unfoldr` is useful for deforestation but only together with `foldr`. With our method we can remove a lot more intermediate lists since we use `destroy` as a good consumer and not (the more restricted) `foldr`.

7 Conclusion and future work

In this paper we have investigated an alternative, less well known technique for shortcut fusion which we call the destroy/unfoldr rule. We have shown that, despite its simplicity, it can tackle problems which have shown to be rather difficult to handle. These problems are fusing functions with accumulating parameters and removing all intermediate lists from zip-like functions.

We have made a prototype implementation using the rules pragma of the Glasgow Haskell Compiler [PJTH01]. The rules pragma allows the programmer to specify left-to-right rewrite rules which the compiler will apply on the program whenever it can. This has allowed us to verify that the transformation presented in this paper works for small examples. More work is needed for the implementation to scale up and we believe that the techniques developed for the foldr/build rule can be used for that.

An interesting path of future work is to see how the foldr/build rule and the destroy/unfoldr rule can cohabit. One way would be to let the compiler choose among several different implementations of a function in order to maximise fusion.

Acknowledgements.

I would like to thank Janis Voigländer for his many insightful comments on this paper and for discussing how his work related to mine. I am very much indebted to Jörgen Gustavsson for suggesting many improvements and for helping out with the examples on the space behaviour of shortcut fusion. My supervisor David Sands helped a lot to improve the presentation of this paper. Finally I thank the referees for their many valuable comments and suggestions for improvement.

S.D.G.

References

[CDPR99] Loic Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. How to deforest in accumulative parameters? Technical Report 3608, INRIA, January 1999.

[Chi92] W. Chin. Safe Fusion for Function Expressions. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 11–20, San Francisco, California, June 1992.

[Chi99] Olaf Chitil. Type inference builds a short cut to deforestation. In *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming*, volume 34 of *ACM Sigplan Notices*, pages 249–260, 1999.

[Feg96] L. Fegaras. Using the parametricity theorem for program fusion. Technical Report 96-001, Oregon Graduate Institute, 1996.

[FSZ94] L. Fegaras, T. Sheard, and T. Zhou. Improving programs which recurse over multiple inductive structures. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Orlando, Florida, June 1994.

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

[GJ98] Jeremy Gibbons and Geraint Jones. The under-appreciated unfold. In *Proceedings 3rd ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'98, Baltimore, MD, USA, 26–29 Sept 1998*, pages 273–279. ACM Press, New York, 1998.

[GLJ93] A. Gill, J. Launchbury, and S. Peyton Jones. A Short Cut to Deforestation. In *Functional Programming and Computer Architecture*, pages 223–232, Copenhagen, Denmark, June 1993.

[GS01] Jörgen Gustavsson and David Sands. Possibilities and limitations of call-by-need space improvement. In *Proceeding of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP'01)*, pages 265–276. ACM Press, September 2001.

[Ham01] G.W. Hamilton. Extending higher order deforestation: Transforming programs to eliminate even more trees. In *Proceedings of the Third Scottish Functional Programming Workshop*, Stirling, Scotland, August 2001.

[HIT96] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. An Extension of the Acid Rain Theorem. In *2nd Fuji International Workshop on Functional and Logic Programming*, pages 91–105, Shonan Village, Japan, November 1996. World Scientific.

[JH99a] S. Peyton Jones and J. Hughes. Haskell 98 library report, February 1999.

[JH99b] S. Peyton Jones and J. Hughes. Haskell 98 report, February 1999.

[Joh01] Patricia Johann. Short cut fusion: Proved and improved. In W. Taha, editor, *Proceedings of the Second International Workshop on Semantics, Applications, and Implementation of Program Generation*, volume LNCS 2196 of *Lecture Notes in Computer Science*, Florence, Italy, September 6 2001. Springer.

[LKS00] J. Launchbury, S. Krstic', and T. E. Sauerwein. Zip Fusion with Hyperfunctions. Oregon Graduate Institute, 2000.

[Mar96] Simon Marlow. *Deforestation for Higher-Order Functional Programs*. PhD thesis, University of Glasgow, 1996.

[MFP91] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In John Hughes, editor, *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA)*, volume 523 of *LNCS*. Springer-Verlag, 1991.

[PJS98] S. Peyton Jones and A. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1–3):3–47, 1998.

[PJTH01] S. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Haskell Workshop*, 2001.

[San98] D. Sands. Improvement theory and its applications. In A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute, pages 275–306. Cambridge University Press, 1998.

[SF93] T. Sheard and L. Fegaras. A Fold for All Seasons. In *Sixth Conference on Functional Programming Languages and Computer Architecture*, pages 233–242, Copenhagen, June 1993.

[TM95] A. Takano and E. Meijer. Shortcut deforestation in calculational form. In Simon L Peyton Jones, editor, *Functional Programming and Computer Architecture*, San Diego, 1995. ACM.

[VK01] J. Voigtländer and A. Kühnemann. Composition of functions with accumulating parameters. Technical Report TUD-FI01-08, Dresden University of Technology, 2001.

[Voi02] J. Voigtländer. Concatenate, reverse and map vanish for free. In *Proceeding of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP'02)*. ACM Press, 2002.

[Wad84] P. Wadler. Listlessness is Better than Laziness: Lazy Evaluation and Garbage Collection at Compile-time. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, Austin, Texas, 1984.

[Wad87] Philip Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *14'th ACM Symposium on Principles of Programming Languages*, Munich, Germany, January 1987.

[Wad89] Philip Wadler. Theorems for free! In *4'th International Conference on Functional Programming and Computer Architecture*, London, September 1989.

[Wad90] Philip Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

A Shortcut fusion is not an improvement

Using parametricity to prove the correctness of an optimisation is a rather weak ground since it only tells us that the expressions will compute the same values before and after transformation. But when we are dealing with transformations we want to know if the transformation actually improves the program or not. A reasonable thing to expect is that the transformation "improves" programs in some improvement theory [San98]. We will here give indications that neither the foldr/build rule nor the destroy/unfoldr rule are space improvements [GS01]. In the examples we will assume the semantics used in the paper by Gustavsson and Sands.

A.1 The foldr/build rule increases sharing

First we will give an example showing that the foldr/build rule can increase sharing in a program. When sharing is increased, bits of memory is retained longer and can therefore result in higher memory demands of the program. Consider the following functions:

```
ones = build (\c n -> let l = c 1 l in l)
map f xs = build (\c n ->
                    foldr (\x xs -> f x 'c' xs) n xs)
```

Suppose we have the following expression:

```
map square ones
```

Each element in the resulting list will be computed separately. Now, suppose we inline the definitions and apply foldr/build fusion. After some β-reductions we will then end up with the following expression:

```
build(\c n -> let l = c (f 1) l in l)
```

We can now see that in the computation `f 1` is computed only once and shared among the elements of the list.

It should be noted that in the original paper on the foldr/build rule [GLJ93], `repeat` was given a definition similar to `ones`. Using that definition will probably lead to increased sharing when the foldr/build rule is used.

A.2 The destroy/unfoldr rule loses sharing

Next, we turn to the destroy/unfoldr rule. Since it is the dual of the foldr/build rule we might expect examples where it can decrease the sharing in a program. This turns out to be the case. Consider the following functions:

```
foo xs = destroy bar xs
  where bar psi xs =
          case psi xs of
            Just (a,ys) -> 1
            Nothing -> case psi xs of
                         Nothing -> 1
                         Just (b,zs) -> 1

traverse [] = Nothing
traverse (x:xs) = traverse xs
```

`foo` is a rather strange looking function but it serves its purpose in the example. The key thing to note is that it performs the call `psi xs` twice.

Now, suppose we have the following expression:

```
foo (unfoldr traverse biglist)
```

where `biglist` is some arbitrary big list. What will happen is that `foo` will try to inspect its list. When doing so `unfoldr traverse biglist` will be evaluated to the empty list. This is done once and for all since it will occur as the argument `xs` in the local function `bar`. Lazy evaluation will make sure that the computation of `xs` inside `bar` is shared with `foo`.

Now, let us see what happens when we transform our expression. We begin by inline the definition of `foo`.

```
destroy bar (unfoldr traverse biglist)
  where bar = ...
```

We have now an opportunity to apply the destroy/unfoldr rule. Doing so and inlining the definition of `bar` will make us end up with:

```
case traverse biglist of
  Just (a,ys) -> 1
  Nothing -> case traverse biglist of
               Nothing -> 1
               Just (b,zs) -> 1
```

In this final expression `traverse biglist` is performed twice. We have thus lost sharing.