Deforestation

Vincent St-Amour

April 26, 2012

1 Functional Programming is Nice

Why do we like functional programming?

Well, one of the reasons *I* like functional programming is that it allows me to write programs that look like the problem they're solving. For example, say I want to:

sum the squares of all numbers from 1 up to n

I want to write:

```
sum_of_squares :: Int -> Int
sum_of_squares n = sum (map square (upto 1 n))
```

not:

```
sum := 0;
for i := 1 to n do
    sum := sum + square(i);
```

The original problem statement is in terms of *sum*, *square* and *from* ... *up to*, not *for* and assignment. I want my program to be the same.

However, the second program is *much* more efficient than the first one. Most importantly, even though all we want is to go from a number (n) to another number (the result), the program ends up allocating two intermediate *lists* along the way: one for the result of `upto`, and one for the result of `map`.

Intermediate lists are inefficient for multiple reasons.

- They take up space; the first program takes $O(n)$ space, while the second takes $O(1)$.
- Each cons cell of each list needs to be allocated, which takes time ($O(n)$ per traversal), and increases GC pressure.
- Each list needs to be traversed when it gets consumed (more time).

So, what we *really* want is to write the first program, and have the compiler *automatically* turn it into the second (or something morally equivalent).

This is where *deforestation* comes in. Deforestation is a transformation that transforms programs written using list combinators (such as `map`, `foldr`, etc.) and eliminates intermediate lists.

Deforestation is a very general technique, and it can apply beyond lists, to trees (which is where the name *deforestation* comes from) and other intermediate data structures. It also applies beyond list combinators to list comprehensions and more. We will ignore all that, and focus on lists and list combinators. Most of the techniques we will see can easily be extended to work on other data structures and other modes of operation.

Before we dive into the meat of the topic, let's outline the classes of list combinators we want our "deforester" to handle:

- Producers (`unfoldr`)
- Consumers (`foldr`)
- Transformers (`map`)
- Selectors (`filter`), no one-to-one mapping from inputs to outputs
- using accumulators (`foldl`)
- traversing multiple lists at once (`zip`)
- traversing the same list multiple times (`average`)

Most of these functions should be familiar, but just in case, here are their definitions:

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

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ z []     =  z
foldr f z (x:xs) =  f x (foldr f z xs)

map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

```
filter :: (a -> Bool) -> [a] -> [a]
filter pred []      = []
filter pred (x:xs)
  | pred x          = x : filter pred xs
  | otherwise       = filter pred xs

foldl :: (a -> b -> a) -> a -> [b] -> a
foldl _ z []     = z
foldl f z (x:xs) = foldl f (f z x) xs

zip :: [a] -> [b] -> [(a,b)]
zip (a:as) (b:bs) = (a,b) : zip as bs
zip _      _      = []

average :: [Int] -> Int
average xs = (sum xs) `div` (length xs)
```

`unfoldr` is not as well-known as the others, so let's see it in action by implementing `upto` in terms of it:

```
upto :: Int -> Int -> [Int]
upto from to = unfoldr (\ s -> if s > to
                               then Nothing
                               else Just (s, s + 1))
                       from
```

2 Timeline

The desire to program in a functional style while achieving imperative-level performance has been around for about as long as functional programming itself. For the rest of the lecture, we will trace the history of deforestation, pausing to study the most important[1] milestones along the way.

We will focus on three chunks of work, all of which led to their author's dissertation:

1. Philip Wadler introduces deforestation in the early 80s.
2. Andy Gill[2] brought deforestation to the mainstream (i.e. GHC) in the early 90s.
3. Duncan Coutts[3] introduced *stream fusion*, which addresses the limitations of Gill's work, in the late 00s.

[1]Disclaimer: most important *from the perspective of compiler optimization*, as determined by a guy who has read on the topic for two weeks (i.e. *not* an expert). Any omissions and/or misrepresentations are accidental.

[2]The faculty member at the University of Kansas, *not* the guitarist for Gang of Four.

[3]The consultant at Well-Typed, *not* the bassist for Our Lady Peace.

3 1976-1981: Prehistory

Before deforestation, multiple techniques had been proposed to eliminate intermediate data structures from functional programs. Here's a brief overview of some of the most influential.

The *fold-unfold* method Burstall and Darlington[2] introduce the *fold-unfold* method, a small set of semantics-preserving transformations that, when applied to functional programs, make them more efficient. The rules, while straightforward, must be applied by a human and are not easily automatable.

Backus's FP In his Turing Award lecture[4], John Backus introduces *function-level* programming and the FP system. FP has been described as a cross between functional programming and APL. FP came with an *algebra* of programs, that allowed programs to be manipulated while preserving correctness. Using this algebra, it is possible to show that eliminating intermediate data structures preserves correctness.

Lazy evaluation Lazy evaluation[1] has been considered as a solution to the inefficiencies of the functional style. Lazy evaluation sometimes eliminates the need for intermediate data structures; their elements get computed on demand, solving space issues. This does not always work, though (e.g. `average`). Furthermore, lazy evaluation does not help to fix the time inefficiencies of the functional style; worse, it often imposes an additional overhead over strict evaluation.

4 1981: Deforestation

Inspired by Burstall's work, Philip Wadler set out to develop a technique that would *automatically* eliminate intermediate data structures from functional programs. He dubbed this technique deforestation.

Wadler identifies a core set of list operators that can be used to express a large class of list computations: `map`, `red`(uce) and `gen`(erate). `red` is very similar to `foldr`[4], so we'll just use `foldr` instead.[5] `gen` serves the same purpose as `unfoldr`, but the two differ enough that we'll stick with `gen`. List generators are not as widely agreed upon as list consumers, anyway.

```
gen :: (a -> Bool) -> (a -> b) -> (a -> a) -> a -> [b]
gen stop yield next seed =
  if stop seed
  then []
  else (yield seed) : (gen stop yield next (next seed))
```

[4]`red` is in fact much closer to `foldr` than to the modern `reduce`

[5]This involves converting Wadler's rules and examples a bit. Any errors are mine.

`gen` differs from `unfoldr` in the following ways: it uses an explicit predicate (its first argument) to determine when to stop, whereas `unfoldr` uses a `Maybe` type, and `gen` uses two separate functions (its second and third arguments) to compute the next value in the list and the next value of the seed, respectively, whereas `unfoldr` uses a single function that returns a pair.

As a simple example, here is the `upto` function defined, using `gen`:

```
upto :: Int -> Int -> [Int]
upto from to = gen (> to) id (+ 1) from
```

Wadler then defines a set of rewrite rules that operate on combinations of these operators. These rules identify specific patterns of list operators that create then consume intermediate lists, then merges the operations to eliminate the need for intermediate lists.

map-map

```
map f (map g xs)  ⇒  map h xs
                       where h x = f g x
```

foldr-map

```
foldr f a (map g xs)  ⇒  foldr h a xs
                           where h x a = f (g x) a
```

map-gen

```
map f (gen stop yield next b)  ⇒  gen stop h next b
                                    where h b = f (yield b)
```

foldr-gen

```
foldr f a (gen stop yield next b)  ⇒  h a b
                                        where h a b =
                                          if stop b
                                          then a
                                          else h (f (yield b) a)
                                                 (next b)
```

The deforestation algorithm basically consists of applying these rules over and over, interspersed with inlining steps to expose more deforestation opportunities.

Let's observe deforestation in action, using our sum of squares example:

```
sum (map square (upto 1 n))
```

we inline the definition of `sum`:

```
foldr (+) 0 (map square (upto 1 n))
```

we apply the *foldr-map* rule, eliminating one of the intermediate lists:

```
foldr h1 0 (upto 1 n)
    where h1 x a = (square x) + a
```

we inline the definition of `upto`:

```
foldr h1 0 (gen (> n) id (+ 1) 1)
    where h1 x a = (square x) + a
```

we apply the *foldr-gen* rule, eliminating the other intermediate list:

```
h2 0 1
    where h2 a b = if (> n) b
                   then a
                   else h2 (h1 a (id b)) ((+ 1) b)
```

finally, a bit of inlining and cleanup:

```
h2 0 1
    where h2 a b = if b > n
                   then a
                   else h2 (a + b * b) (b + 1)
```

which is equivalent[6] to the fast program from the introduction. We win.

The above rules cover a large class of list operations, but not all the ones we set out to handle in the introduction. To support functions that traverse a given list multiple times (e.g. `average`), we need an extra rule, that collapses two traversals into one:

parallel foldr

```
(foldr f a xs, foldr g b xs)  ⇒  foldr h (a, b) xs
                                   where h x (a, b) = (f x a, g x b)
```

But other kinds of operations, such `filter`, cannot be expressed in this system. Referring back to our goals from the introduction, we get:

	unfoldr	foldr	map	filter	foldl	zip	average
Wadler 81	X	X	X				X

Wadler implemented a prototype transformer[7] that used this technique, showing its feasibility. It's unclear whether that prototype was integrated to a compiler or not.

Let's wrap up our discussion of this work.

[6]The equivalence of tail-recursion and iteration had been shown by Steele (Wadler's co-advisor) four years earlier[3].

[7]In Lisp.

Advantages

- *Automatic*: deforestation does not require programmer input, and is therefore suitable for inclusion in a compiler.
- *Source-to-source*: all the transformations take valid programs in the source language to other (faster) valid programs in the source language. This makes easy to integrate to a compiler pipeline, and makes it possible to apply further optimizations to the output of deforestation. It also means that other optimizations can potentially introduce new opportunities for deforestation.
- *Simple*: each rule is easy to understand and obviously correct.

Disadvantages

- *One optimization = one rule*: if we want to add new list operators, we need to add a new rule for each combination of the new operator with each of the old ones. This does not scale.
- *Limited*: a lot of interesting list computations cannot be covered using this framework. We cannot eliminate intermediate lists in these cases.

5 Interlude: Where are the Types?

I'm glad you asked!

We haven't used types explicitly so far, but they are necessary to ensure correctness. Consider the following program, and assume we're in an untyped setting:

```
map 3 (gen (\ x -> True) id id 0)
```

`gen` stops right away, producing the empty list. `map` gets called, and before it does anything, checks whether `3` is a function (it's not) and errors. After optimization, this program becomes:

```
gen (\ x -> True) h id 0
  where h b = 3 (id b)
```

In this case, `h` never gets called, so `3` never gets checked for functionness, and the program returns the empty list. We just turned an erroring program into one that runs to completion. This is bad. More generally, deforestation can potentially move errors around, which makes debugging harder. This is also bad. Types prevent all this from happening.

6 1984-1990: The Listless Transformer

Throughout the 80s, Wadler kept working on deforestation. In 1984, he introduced a new deforestation technique, which he dubbed the *listless transformer*[6]. This transformer would take a program written using list operators, and would transform it into a program for a *listless machine*, that is, a first-order, constant-space assembly-level program.

The listless transformer can deforest a larger class of programs than Wadler's original approach, but has several drawbacks. First, the transformer is fairly complicated; it lacks the simplicity and "obvious correctness" of the original deforestation algorithm.

Second, the transformer does not produce valid source programs; it produces output for a special machine. This makes the transformer hard to fit in a compiler pipeline, and makes further optimizations on the transformer's output awkward. In a subsequent[8] paper[8], Wadler explains how to integrate listless machines with graph reducers, but this is more of a Foreign Function Interface than a true integration.

Finally, the listless transformer imposes restrictions on its inputs that limit its applicability, but are necessary to ensure termination of the transformer.

7 1993: build-foldr Fusion

Andy Gill's doctoral work introduces another approach to deforestation, *build-foldr fusion*, which performs a range of optimizations as broad as the listless transformer, but addresses its drawbacks.

Like the original deforestation algorithm, build-foldr fusion is source-to-source and rule-based. In fact, it is based on a single rule:

foldr-build

`foldr k z (build g)` $\Rightarrow$ `g k z`

Cool, what's build?

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

`build` itself is not especially interesting. Most of the action happens in `g`. `g` is whatever list producer we want, abstracted over `:` (i.e. cons) and `[]` (i.e. nil). That is, instead of calling cons and nil directly, it calls the cons and nil it

[8]Literally the next one in the sequence.

gets as arguments. `build` simply passes `g` the usual cons and nil.

To make this concrete, let's see what our `upto` function would look like in this style:

```
upto :: Int -> Int -> [Int]
upto from to = if from > to
               then []
               else from : upto (from + 1) to
```

if we abstract over `:` and `[]`, we get:

```
upto' :: Int -> Int -> (a -> b -> b) -> b -> b
upto' from to =
  \ cons' nil' -> if from > to
                  then nil'
                  else cons' from (upto' (from + 1) to)
```

It should be obvious from these definitions that we can derive `upto` using `build` and `upto'`. Good, but why bother? The answer comes from `foldr`:

```
foldr (+) 0 (1 : 2 : 3 : 4 : [])
```

is equivalent to:

```
1 + 2 + 3 + 4 + 0
```

In a sense, `foldr` replaces the conses in its list argument with its function argument, and the nil of its list argument with its nil argument. `foldr` *undoes* what `build` does! The two cancel out nicely, which is precisely what the foldr-build rule is all about.

Let's look at an example:

```
sum (upto 1 n)
```

we inline the definitions of `sum` and `upto`:

```
foldr (+) 0 (build (upto' 1 n))
```

we apply the build-foldr rule:

```
(upto' 1 n) (+) 0
```

to better see what's going on, let's inline the definition of `upto'`:

```
h 0
    where h from = if from > n
                   then 0
                   else from + (h (from + 1))
```

We get the code we want. Groovy.

Now let's go back to the type of `build`:

$$\forall\alpha.\,(\forall\beta.\,(\alpha \to \beta \to \beta) \to \beta \to \beta) \to [\alpha]$$

This type may look more complicated than necessary, but it is in fact necessary for correctness. Let's assume we use a more permissive type for `build`:

$$\forall\alpha\beta.\,((\alpha \to \beta \to \beta) \to \beta \to \beta) \to [\alpha]$$

Bad things can happen:

```
foldr (+) 0 (build (\ c n -> 3 : []))
```

This typechecks just fine with the permissive type. α is `Int` and β is $[\alpha]$. Using the build-foldr rule, this gets transformed to:

```
(\ c n -> 3 : []) (+) 0
```

That can't be right, the two expressions don't reduce to the same thing. They don't even have the same type!

`3` $\neq$ `[3]`

That's where the more restricted type saves us. It *forces* `build`'s argument to *only* use the cons and nil it gets from `build` to construct its result. It can't use a cons and nil it smuggled from somewhere else, as in our example. Thanks to the magic of Free Theorems, we know that this can't happen.

However, this is a rank-2 type, which is *not* supported by Hindley-Milner. Oops. To make this work in Haskell, Gill et al. use `build` internally, but do not provide it to the use, who could potentially abuse it. With a more powerful type system, though, it would just work.

Now let's scale that up a bit:

```
sum (map square (upto 1 n))
```

as before, we inline:

```
foldr (+) 0 (map square (build (upto' 1 n)))
```

and we're stuck.

How can we extend this technique so that it works on `map` and friends? Easy, we write them using `build` and `foldr`!

```
map :: (a -> b) -> [a] -> [b]
map f xs = build (\ c nil -> foldr (\ x a -> c (f x) a) nil xs)

filter :: (a -> Bool) -> [a] -> [a]
filter f xs = build (\ c nil ->
    foldr (\ x a -> if f x then c x a else a) nil xs)
```

And so on.

Let's see if that fixes our problem.

```
foldr (+) 0 (map square (build (upto' 1 n)))
```

we inline `map`:

```
foldr (+) 0
    (build (\ c nil -> foldr (\ x a -> c (square x) a) nil
                             (build (upto' 1 n))))
```

we eliminate the inner intermediate list:

```
foldr (+) 0
    (build (\ c nil -> (upto' 1 n)
                         (\ x a -> c (square x) a) nil))
```

and the outer intermediate list:

```
(\ c nil -> (upto' 1 n) (\ x a -> c (square x) a) nil) (+) 0
```

clean up a bit:

```
(upto' 1 n) (\ x a -> (square x) + a) 0
```

and if we keep inlining, we'll get the fast program from the introduction. We're done.

Going back to our scoreboard, let's see how build-foldr fares:

	`unfoldr`	`foldr`	`map`	`filter`	`foldl`	`zip`	`average`
Wadler 81	X	X	X				X
Gill 93	X	X	X	X			X

This is better than the original deforestation algorithm, but there are still cases that we can't handle. We can express `foldl` in terms of `foldr`, but that just moves the intermediate list to the accumulator position, so we don't gain anything. In the case of `zip`, build-foldr fusion will eliminate one of the lists that `zip` consumes, but can't do anything about the second.

Gill et al. implemented their technique in GHC, and it worked really well in practice. Great success!

Let's wrap up.

Advantages

- *Simple, source-to-source and automatic*: it has all the things we liked about the original deforestation algorithm.
- *No limitations on inputs*: unlike the listless transformer, build-foldr fusion does not impose limitations on its inputs; it can operate on the entire Haskell language. Of course, this does not mean that it will *optimize* all these programs, but at least it won't break or go off in an infinite loop.
- *Single rule*: unlike the original deforestation algorithm, a single rule suffices. When we add new list operations, as long as we express them using `build` and `foldr`, we're good. That's an $O(1)$ amount of work, no need to consider all the combinations with the existing operations.

Disadvantages

- *Limitations*: it can't handle functions that use acculumators (`foldl`) or functions that consume multiple lists (`zip`).

8 1993-2007: Attempts at Improving build-foldr

Following Andy Gill's work, a whole cottage industry of new deforestation approaches appeared, all of them trying to cover the cases left out by build-foldr. Most of that work was theoretical, a lot of it was based on category theory, and almost none of it had any impact on practice.

Most of the proposed approaches were either complicated, or did not cover some cases that build-foldr did, or were not suited for integration in a compiler. As a result, build-foldr remained the favored approach in actual compilers.

9 2007: Stream Fusion

With that context, we arrive at the last system we'll examine: *stream fusion.* Like others in the preceding two decades, its goal was to cover the cases that were missing from Gill's work. Unlike the others, it succeeded and at the same time stayed simple and useful in practice.

Stream fusion has a similar structure to build-foldr fusion: a single fusion rule eliminates matching pairs of constructor and destructor, and list operations are expressed in terms of this constructor and destructor. The constructor/destructor pair used by stream fusion is `stream` and `unstream`:

```
data Stream a = exists s. Stream (s -> Step a s) s
data Step a s = Done | Yield a s | Skip s
```

```
stream :: [a] -> Stream a
stream xs_0 = Stream next xs_0
  where next []       = Done
        next (x : xs) = Yield x xs

unstream :: Stream a -> [a]
unstream (Stream next_0 s_0) = unfold s_0
    where unfold s = case next_0 s of
        Done       -> []
        Skip s'    -> unfold s'
        Yield x s' -> x : unfold s'
```

Streams have some abstract state type and a *stepper* function that advances the stream from one state to the next and may yield values. The `stream` constructor yields values from its input list until it runs out. The `unstream` destructor consumes its input stream until it runs out, building a list from the yielded values and discarding the `Skip`s along the way.

The existential in the `Stream` type is necessary for correctness, for similar reasons as the nested quantification in the type of `build`. To preserve the invariants on which deforestation depends, programmers should not be able to forge stream states.

We can see that `stream` and `unstream` cancel out. This is expressed in the *stream-unstream* rule, the fusion rule used by stream fusion:

stream-unstream

`stream (unstream s)` $\Rightarrow$ `s`

Let's see what `map` and `filter` would look like in this setting:

```
map :: (a -> b) -> [a] -> [b]
map f = unstream . map_s f . stream

map_s :: (a -> b) -> Stream a -> Stream b
map_s f (Stream next_0 s_0) = Stream next s_0
  where next s = case next_0 s of
      Done       -> Done
      Skip s'    -> Skip s'
      Yield x s' -> Yield (f x) s'

filter :: (a -> Bool) -> [a] -> [a]
filter p = unstream . filter_s p . stream
```

```
filter_s :: (a -> Bool) -> Stream a -> Stream a
filter_s p (Stream next_0 s_0) = Stream next s_0
  where next s = case next_0 s of
      Done                   -> Done
      Skip s'                -> Skip s'
      Yield x s' | p x       -> Yield x s'
                 | otherwise -> Skip s'
```

They become simple wrappers around functions that operate on streams. It's worth noting that neither `map_s` nor `filter_s` is recursive. Each time they get called, the stream steps once. This is a property we will want of all our stream processing functions (steppers). In order for the desired computation to be performed, there needs to be a consumer "pulling" at the end, in this case `unstream`.

Using this definition, we can deforest map of map:

```
map f . map g

unstream . map_s f . stream . unstream . map_s g . stream

unstream . map_s f . map_s g . stream
```

The intermediate list goes away as planned.

`foldr` is not primitive anymore, so let's define it:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z xs = foldr_s f z (stream xs)

foldr_s :: (a -> b -> b) -> b -> Stream a -> b
foldr_s f z (Stream next s_0) = go s_0
  where go s = case next s of
      Done       -> z
      Skip s'    -> go s'
      Yield x s' -> f x (go s')
```

Like `unstream`, `foldr` consumes streams, and is allowed to be recursive, since it needs to "pull" all the values from its input.

We don't have a primitive list constructor anymore, so let's see what `upto` would look like:

```
upto :: Int -> Int -> [Int]
upto from to = unstream . upto_s from to

upto_s :: Int -> Int -> Stream Int
upto_s from to = Stream next from
  where next n | n > h     = Done
               | otherwise = Yield n (n + 1)
```

Pretty simple, it just creates a stream. Since we can now express `foldl` and `zip`, let's do them too:

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f z xs = foldl_s f z (stream xs)

foldl_s :: (b -> a -> b) -> b -> Stream a -> b
foldl_s f z (Stream next s_0) = go z s_0
  where go z s = case next s of
      Done       -> z
      Skip s'    -> go z s'
      Yield x s' -> go (f z x) s'

zip :: [a] -> [b] -> [(a,b)]
zip xs ys = unstream . zip_s (stream xs) (stream ys)

zip_s :: Stream a -> Stream b -> Stream (a,b)
zip_s (Stream next_a s_a0) (Stream next_b s_b0) =
    Stream next (s_a0, s_b0, Nothing)
  where next (s_a, s_b, Nothing) =
            case next_a s_a of
                Done         -> Done
                Skip s_a'    -> Skip (s_a', s_b, Nothing)
                Yield a s_a' -> Skip (s_a', s_b, Just a)
        next (s_a', s_b, Just a) =
            case next_b s_b of
                Done         -> Done
                Skip s_b'    -> Skip (s_a', s_b', Just a)
                Yield b s_b' -> Yield (a,b) (s_a', s_b', Nothing)
```

`foldl` is actually quite simple. It's allowed to be recursive, so we can have the accumulators we want. `zip` is a bit more involved. The stream it produces needs to keep track of the state of both its inputs and in addition, it keeps a "buffer" for an `a`. Since `zip_s` is a stepper, it's not allowed to be recursive; it needs to produce a value for each action it takes. However, in order to yield a value, both its input streams need to yield a value. To make it all work, we encode a simple state machine. Every time the stepper function gets called, it makes a transition. In first state, the stepper tries to read from its first input stream. If it yields a value, the stepper stashes it away, then skips and goes into the second state. In the second state, the stepper tries to read from its second input stream. If it yields a value, the stepper combines it with its stashed value and yields the result, resets its buffer, then goes back to the first state.

Ok, so things look good, we can eliminate intermediate lists. But wait, we're not done yet! We've only traded intermediate lists for intermediate stream structures and intermediate stream states. We haven't gained anything on the allocation front!

Fear not! We're now in a position where standard compiler optimization techniques can eliminate the remaining allocation. Let's look at an example:

```
foldr_s (+) 0 . map_s square . stream xs
```

First, let's inline the definitions:

```
go_foldr 0 xs
  where go_foldr z s = case next_map s of
           Done       -> z
           Skip s'    -> go_foldr z s'
           Yield x s' -> go_foldr (x + z) s'
        next_map xs = case next_stream xs of
           Done       -> Done
           Skip s'    -> Skip s'
           Yield x s' -> Yield (square x) s'
        next_stream xs = case xs of
           []         -> Done
           (x : xs')  -> Yield x xs'
```

Since all steppers functions are non-recursive, we can fuse them via inlining. Let's fuse `next_map` and `next_stream`:

```
go_foldr 0 xs
  where go_foldr z s = case next_map s of
           Done       -> z
           Skip s'    -> go_foldr z s'
           Yield x s' -> go_foldr (x + z) s'
        next_map xs = case case xs of
                             []        -> Done
                             (x : xs') -> Yield x xs' of
           Done       -> Done
           Skip s'    -> Skip s'
           Yield x s' -> Yield (square x) s'
```

Then, we can use the *case-of-case* transformation[9], which pushes the outer *case* in the alternatives of the inner *case*:

[9]Your compiler does that, right? If not, why are you bothering with stream fusion? You've got lower hanging fruit than that to pick.

```
go_foldr 0 xs
  where go_foldr z s = case next_map s of
           Done       -> z
           Skip s'    -> go_foldr z s'
           Yield x s' -> go_foldr (x + z) s'
        next_map xs = case xs of
           []        -> case Done of
                          Done       -> Done
                          Skip s'    -> Skip s'
                          Yield x s' -> Yield (square x) s'
           (x : xs') -> case Yield x xs'
                          Done       -> Done
                          Skip s'    -> Skip s'
                          Yield x s' -> Yield (square x) s'
```

This trivially rewrites to:

```
go_foldr 0 xs
  where go_foldr z s = case next_map s of
           Done       -> z
           Skip s'    -> go_foldr z s'
           Yield x s' -> go_foldr (x + z) s'
        next_map xs = case xs of
           []        -> Done
           (x : xs') -> Yield (square x) xs'
```

And we've managed to eliminate the residual allocation from the inner part of the example; all traces of streams have disappeared. If we keep applying the same optimizations, we can deforest the rest, too:

```
go_foldr 0 xs
  where go_foldr z s = case s of
           []        -> z
           (x : xs') -> go_foldr ((square x) + z) xs'
```

Note that all the optimizations we applied are standard; none are deforestation-specific.

Ok, let's see what the scoreboard looks like:

	`unfoldr`	`foldr`	`map`	`filter`	`foldl`	`zip`	`average`
Wadler 81	X	X	X				X
Gill 93	X	X	X	X			X
Coutts 07	X	X	X	X	X	X	X

Stream fusion is now implemented in GHC[10], it works well in practice and is responsible for big speedups all over.

Let's wrap up our discussion of stream fusion.

Advantages

- *All of the above*: stream fusion preserves all the nice things about the previous systems.
- *Complete coverage*: and it covers all the cases we set out to cover.

Disadvantages

- *Nothing, really*: stream fusion is the state of the art. The only "shortcoming" I can think of is that it can't apply unless you write programs using list operators; if you insist on writing your list functions recursively, it won't help you. But that's true of all the systems we've seen. Don't do that.

10 Conclusion

Deforestation allows us to have our cake and eat it, too. We can write our programs in a nice, high-level functional style, using all the list operators we want and, using deforestation, our compiler will eliminate the intermediate data structures introduced by the functional style.

The idea of eliminating these intermediate data structures has been around for almost as long as functional programming itself, and automated solutions have been the subject of research for about 30 years. Over time, solutions have increased in applicability and flexibility, and we've reached a point where the vast majority of programs built using list operators can be deforested. Deforestation has made its way into mainstream functional compilers (most notably GHC) and has led to significant speedups in practice.

[10]Replacing build-foldr fusion, I assume.

References

[1] Daniel P. Friedman, David S. Wise, *Cons should not evaluate its arguments.* Automata, Languages and Programming, 1976.

[2] Rod M. Burstall, John Darlington, *A transformation system for developing recursive programs.* Journal of the ACM, 1977.

[3] Guy L. Steele Jr., *Debunking the expensive procedure call myth, or, Procedure call implementations considered harmful, or, LAMBDA: the ultimate goto.* ACM Annual Conference, 1977.

[4] John Backus, *Can programming be liberated from the Von Neumann style?.* Communications of the ACM, 1978.

[5] Philip Wadler, *Applicative style programming, program transformation, and list operators.* Conference on Functional Programming and Computer Architecture, 1981.

[6] Philip Wadler, *Listlessness is better than laziness.* Conference on LISP and Functional Programming, 1984.

[7] Philip Wadler, *Listlessness is Better than Laziness.* Ph.D. dissertation, Carnegie-Mellon University, 1984.

[8] Philip Wadler, *Listlessness is better than laziness II.* Workshop on Programs as Data Objects, 1985.

[9] Andrew Gill, John Launchbury, Simon L Peyton Jones, *A short cut to deforestation.* Conference on Functional Programming and Computer Architecture, 1993.

[10] Andrew Gill, *Cheap Deforestation for Non-Strict Functional Languages.* Ph.D. dissertation, University of Glasgow, 1996.

[11] Duncan Coutts, Roman Leshchinskiy, Don Stewart, *Stream Fusion.* International Conference on Functional Programming, 2007.

[12] Duncan Coutts, *Stream Fusion: Practical Shortcut Fusion for Coinductive Sequence Types.* Ph.D. dissertation, University of Oxford, 2010.