Fusion: Applying Equational Transforms to Simplify Programs

`github.com/ryanorendorff/lc-2017-fusion`

Ryan Orendorff[1]

May 2017

[1]Department of Bioengineering
University of California, Berkeley
University of California, San Francisco

Outline

Motivation: Simple Programs versus Performance

A brief introduction to GHC

List fusion with $foldr/build$

Stream Fusion

Applications of Fusion

Motivation: Simple Programs versus Performance

Common way to process a list: map and fold!

As an example, say we want to square all the elements in a list and then sum the result.

$$process :: [Int] \to Int$$

$$process\ xs = sum \circ map\ sq\ \$\ xs$$

Where we have defined the functions as follows.

$$\begin{aligned} map\ \_\ [\,] &= [\,] \\ map\ f\ (x : xs) &= f\ x : map\ f\ xs \end{aligned}$$

$$sq\ x = x * x$$

Common way to process a list: map and fold!

As an example, say we want to square all the elements in a list and then sum the result.

$$process :: [Int] \to Int$$
$$process\ xs = sum \circ map\ sq\ \$\ xs$$

Where we have defined the functions as follows.

$$\begin{aligned} map\ \_\ [\,] &= [\,] \\ map\ f\ (x : xs) &= f\ x : map\ f\ xs \end{aligned}$$

$$sq\ x = x * x$$

$$foldr :: (a \to b \to b) \to b \to [a] \to b$$
$$\begin{aligned} foldr\ \_\ z\ [\,] &= z \\ foldr\ f\ z\ (x : xs) &= f\ x\ (foldr\ f\ z\ xs) \end{aligned}$$

$$sum = foldr\ (+)\ 0$$

How fast is $process$?

So now that we have our process function, how fast does it run?

$$process :: [Int] \rightarrow Int$$

$$process\ xs = sum \circ map\ sq\ \$\ xs$$

Let's try to process a million elements with our $process$ and $process'$, which uses the standard Prelude sum and map.

$$process\ [0 \mathinner{..} 1,000,000]; process'\ [0 \mathinner{..} 1,000,000]$$

How fast is $process$?

So now that we have our process function, how fast does it run?

$$process :: [Int] \rightarrow Int$$

$$process\ xs = sum \circ map\ sq\ \$\ xs$$

Let's try to process a million elements with our $process$ and $process'$, which uses the standard Prelude sum and map.

$$process\ [0\,..\,1{,}000{,}000];process'\ [0\,..\,1{,}000{,}000]$$

Function	Time (ms)	Memory (MB)
$process$	41.86	265.26
$process'$	25.31	96.65

How fast is $process$?

So now that we have our process function, how fast does it run?

$$process :: [Int] \to Int$$

$$process\ xs = sum \circ map\ sq \ \$\ xs$$

Let's try to process a million elements with our $process$ and $process'$, which uses the standard Prelude sum and map.

$$process\ [0 \mathinner{..} 1{,}000{,}000]; process'\ [0 \mathinner{..} 1{,}000{,}000]$$

Function	Time (ms)	Memory (MB)
$process$	41.86	265.26
$process'$	25.31	96.65

How does the Prelude do better with the same functions?

We can get good performance with manual code

We can try to get better performance by writing our program as a recursive function.

$$process :: [Int] \rightarrow Int$$
$$process\ xs = sum \circ map\ sq\ \$\ xs$$

We can get good performance with manual code

We can try to get better performance by writing our program as a recursive function.

$$process :: [Int] \rightarrow Int$$
$$process\ xs = sum \circ map\ sq\ \$\ xs$$

$$process_{hand} :: [Int] \rightarrow Int$$
$$process_{hand}\ [\,] = 0$$
$$process_{hand}\ (x : xs) = x * x + process_{hand}\ xs$$

We can get good performance with manual code

We can try to get better performance by writing our program as a recursive function.

$$process :: [Int] \to Int$$
$$process\ xs = sum \circ map\ sq\ \$\ xs$$

$$process_{hand} :: [Int] \to Int$$
$$process_{hand}\ [\,] = 0$$
$$process_{hand}\ (x : xs) = x * x + process_{hand}\ xs$$

function	time (ms)
$process$	41.86
$process'$	25.31
$process_{hand}$	26.8

It seems we have matched GHC's performance!

GHC generated the simplified version automatically

Our manual version $process_{hand}$.

$$process_{hand} :: [Int] \rightarrow Int$$

$$\begin{aligned} &process_{hand}\ [\,] = 0 \\ &process_{hand}\ (x : xs) = x * x + process_{hand}\ xs \end{aligned}$$

and when we compile the Prelude defined $process'$, GHC produces

$$\begin{aligned} &processGHC :: [Int] \rightarrow Int \\ &processGHC\ [\,] \qquad\quad = 0 \\ &processGHC\ (x : xs) = x * x + (processGHC\ xs) \end{aligned}$$

GHC generated the simplified version automatically

Our manual version $process_{hand}$.

$$process_{hand} :: [Int] \rightarrow Int$$

$$\begin{aligned}
&process_{hand}\ [\,] = 0 \\
&process_{hand}\ (x : xs) = x * x + process_{hand}\ xs
\end{aligned}$$

and when we compile the Prelude defined $process'$, GHC produces

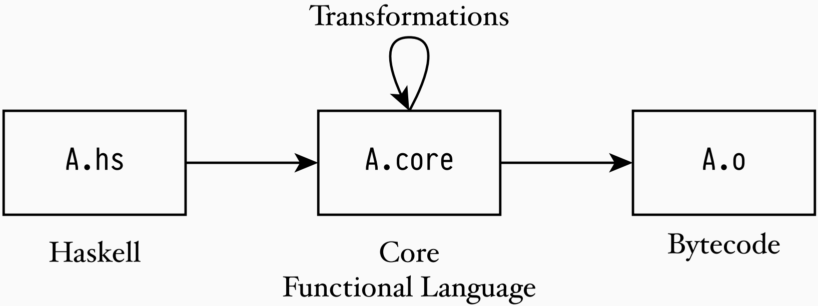
$$\begin{aligned}
&processGHC :: [Int] \rightarrow Int \\
&processGHC\ [\,] \qquad\quad = 0 \\
&processGHC\ (x : xs) = x * x + (processGHC\ xs)
\end{aligned}$$

How can we leverage the compiler to write simple code that is fast?

Table of Contents

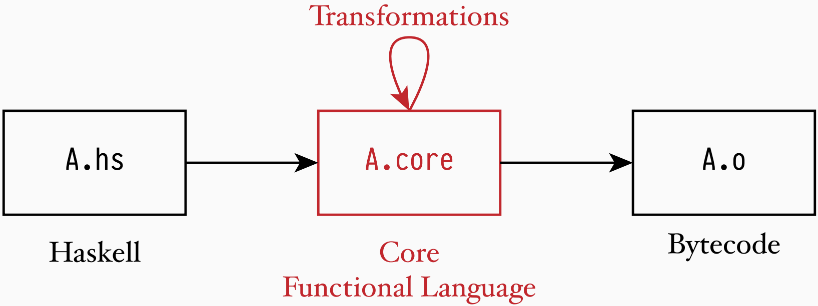
The GHC Compilation Pipeline converts Haskell into an intermediate language and then bytecode

When GHC compiles a Haskell program, it converts the code into an intermediate language called "Core", which is then (eventually) turned into byte code.



The GHC Compilation Pipeline converts Haskell into an intermediate language and then bytecode

When GHC compiles a Haskell program, it converts the code into an intermediate language called "Core", which is then (eventually) turned into byte code.



GHC performs several program transformations on Core to optimize the code

When GHC is given a Core program, it performs several types of transformations on the program.

- Inlining functions

GHC performs several program transformations on Core to optimize the code

When GHC is given a Core program, it performs several types of transformations on the program.

- Inlining functions
- Playing with lambda expressions

GHC performs several program transformations on Core to optimize the code

When GHC is given a Core program, it performs several types of transformations on the program.

- Inlining functions
- Playing with lambda expressions
- Simplifying constant expressions $((x + 8) - 1)$

GHC performs several program transformations on Core to optimize the code

When GHC is given a Core program, it performs several types of transformations on the program.

- Inlining functions
- Playing with lambda expressions
- Simplifying constant expressions $((x + 8) - 1)$
- Reordering case and let expressions

GHC performs several program transformations on Core to optimize the code

When GHC is given a Core program, it performs several types of transformations on the program.

- Inlining functions
- Playing with lambda expressions
- Simplifying constant expressions $((x + 8) - 1)$
- Reordering case and let expressions
- *Applying rewrite rules*

GHC performs several program transformations on Core to optimize the code

When GHC is given a Core program, it performs several types of transformations on the program.

- Inlining functions
- Playing with lambda expressions
- Simplifying constant expressions $((x + 8) - 1)$
- Reordering case and let expressions
- *Applying rewrite rules*
- ...

Rewrite Rules allow us to say two expressions are equivalent

Rewrite rules allow us to replace terms in the program with equivalent terms.

```
{-# RULES "name" [#] forall x. id x = x #-}
```

Rewrite Rules allow us to say two expressions are equivalent

Rewrite rules allow us to replace terms in the program with equivalent terms.

```
{-# RULES "name" [#] forall x. id x = x #-}
```

- "name" is just for us to read when debugging

Rewrite Rules allow us to say two expressions are equivalent

Rewrite rules allow us to replace terms in the program with equivalent terms.

```
{-# RULES "name" [#] forall x. id x = x #-}
```

- "name" is just for us to read when debugging
- [#] represents what phase the rule is applied (phases 4-0)

Rewrite Rules allow us to say two expressions are equivalent

Rewrite rules allow us to replace terms in the program with equivalent terms.

```
{-# RULES "name" [#] forall x. id x = x #-}
```

- "name" is just for us to read when debugging
- `[#]` represents what phase the rule is applied (phases 4-0)
- The `forall` brings a variable into scope

Rewrite Rules allow us to say two expressions are equivalent

Rewrite rules allow us to replace terms in the program with equivalent terms.

```
{-# RULES "name" [#] forall x. id x = x #-}
```

- "name" is just for us to read when debugging
- `[#]` represents what phase the rule is applied (phases 4-0)
- The `forall` brings a variable into scope
- After the period is the what we are saying are equivalent statements.

Rules have some restrictions

Rewrite rules have some gotchas.

- Rules doesn't prevent you from doing something silly

```
{-# RULES "id5" forall x. id x = 5 #-}
```

Rules have some restrictions

Rewrite rules have some gotchas.

- Rules doesn't prevent you from doing something silly

```
{-# RULES "id5" forall x. id x = 5 #-}
```

- The left hand side is only substituted for the right, not the other way around.

```
{-# RULES "id" forall x. id x = x #-}
```

$x \nRightarrow idx$

Rules have some restrictions

Rewrite rules have some gotchas.

- Rules doesn't prevent you from doing something silly

```
{-# RULES "id5" forall x. id x = 5 #-}
```

- The left hand side is only substituted for the right, not the other way around.

```
{-# RULES "id" forall x. id x = x #-}
```

$x \not\Rightarrow idx$

- You can make the compiler go into an infinite loop.

```
{-# RULES "fxy" forall x y. f x y = f y x #-}
```

Rules have some restrictions

Rewrite rules have some gotchas.

- Rules doesn't prevent you from doing something silly

  ```
  {-# RULES "id5" forall x. id x = 5 #-}
  ```

- The left hand side is only substituted for the right, not the other way around.

  ```
  {-# RULES "id" forall x. id x = x #-}
  ```

  $x \nRightarrow idx$

- You can make the compiler go into an infinite loop.

  ```
  {-# RULES "fxy" forall x y. f x y = f y x #-}
  ```

- If multiple rules are possible, GHC will randomly choose one.

We can combine maps to traverse a list once

Let us introduce the following rule about maps.

```
{-# RULES "map/map" forall f g xs.
  map_fuse f (map_fuse g xs) =  map_fuse (f.g) xs #-}
```

We can combine maps to traverse a list once

Let us introduce the following rule about maps.

```
{-# RULES "map/map" forall f g xs.
  map_fuse f (map_fuse g xs) =  map_fuse (f.g) xs #-}
```

$$mapTest :: [Int] \to [Int]$$

$$mapTest\ xs = map\ (+1)\ (map\ (*2)\ xs)$$

$$mapTest_{fuse} :: [Int] \to [Int]$$

$$mapTest_{fuse}\ xs = mapfuse\ (+1)\ (mapfuse\ (*2)\ xs)$$

Our map fusion performs (a bit) better!

We can test our functions on a million elements

$$mapTest\ xs = map\ (+1)\ (map\ (*2)\ xs)$$

$$mapTest_{fuse}\ xs = mapfuse\ (+1)\ (mapfuse\ (*2)\ xs)$$

and find we get a bit better time and space performance.

Function	Time (ms)	Memory (MB)
$mapTest$	26.4	256.00
$mapTest_{fuse}$	17.6	184.00

Through rules, GHC performs fusion

Some of the rules work together to perform *fusion*: to combine terms in such a way as to pass over a data structure once.

In our $process$ function, we create an intermediate list

$$process :: [Int] \rightarrow Int$$

$$process\ xs = sum \circ map\ sq\ \$\ xs$$

whereas our "fused" form did not make any intermediate structure, and used an accumulator instead.

$$process_{hand} :: [Int] \rightarrow Int$$

$$process_{hand}\ [\,] = 0$$
$$process_{hand}\ (x : xs) = x * x + process_{hand}\ xs$$

Table of Contents

foldr / *build* fusion is used to simplify list computations

GHC accomplishes fusion with two functions: foldr and build.

$foldr$ / $build$ fusion is used to simplify list computations

GHC accomplishes fusion with two functions: foldr and build.

$foldr$ combines the elements of a list

$$foldr :: (a \to b \to b) \to b \to [\,a\,] \to b$$
$$foldr\ f\ \ z\ [\,] = z$$
$$foldr\ f\ \ z\ (x : xs) = f\ \ x\ (foldr\ f\ \ z\ xs)$$

foldr / *build* fusion is used to simplify list computations

GHC accomplishes fusion with two functions: foldr and build.

foldr combines the elements of a list

$$foldr :: (a \to b \to b) \to b \to [a] \to b$$
$$foldr\ f\ z\ [\,] = z$$
$$foldr\ f\ z\ (x : xs) = f\ x\ (foldr\ f\ z\ xs)$$

while *build* builds up a list from a generating function.

$$build :: \forall a.(\forall b.(a \to b \to b) \to b \to b) \to [a]$$
$$build\ g = g\ (:)\ [\,]$$

$foldr$ / $build$ fusion is used to simplify list computations

GHC accomplishes fusion with two functions: foldr and build.

$foldr$ combines the elements of a list

$$
\begin{aligned}
&foldr :: (a \to b \to b) \to b \to [a] \to b \\
&foldr\ f\ z\ [\,] = z \\
&foldr\ f\ z\ (x : xs) = f\ x\ (foldr\ f\ z\ xs)
\end{aligned}
$$

while $build$ builds up a list from a generating function.

$$
\begin{aligned}
&build :: \forall a.(\forall b.(a \to b \to b) \to b \to b) \to [a] \\
&build\ g = g\ (:)\ [\,]
\end{aligned}
$$

$$
\begin{aligned}
&build1\ l \equiv [1, 2, 3] \\
&\quad \textbf{where} \\
&\qquad l\ cons\ nil = 1\ `cons`\ (2\ `cons`\ (3\ `cons`\ nil))
\end{aligned}
$$

The *foldr*/*build* rule removes intermediate fold/build pairs

To remove intermediate data structures (those created by *build*), we eliminate *foldr*/*build* pairs with a rule.

```
{-# RULES
"foldr/build"
∀ f z (g :: ∀ b. (a -> b -> b) -> b -> b).
foldr f z (build g) = g f z #-}
```

$$
\begin{aligned}
&\mathit{foldr}\ (+)\ 0\ (\mathit{build}\ l) \equiv l\ (+)\ 0 \equiv 1 + (2 + (3 + 0))\\
&\quad \textbf{where}\\
&\qquad l\ \mathit{cons}\ \mathit{nil} = 1\ \text{`}\mathit{cons}\text{`}\ (2\ \text{`}\mathit{cons}\text{`}\ (3\ \text{`}\mathit{cons}\text{`}\ \mathit{nil}))
\end{aligned}
$$

We need a few extra rules to convert maps into fold/builds

To convert our definition of maps into a fold/build pair, we need the following helper function.

$$
\begin{aligned}
&mapFB :: (elt \to lst \to lst) \to (a \to elt) \to a \to lst \to lst \\
&mapFB\ c\ f = \lambda x\ ys \to c\ (f\ x)\ ys
\end{aligned}
$$

We need a few extra rules to convert maps into fold/builds

To convert our definition of maps into a fold/build pair, we need the following helper function.

$$
\begin{aligned}
&\mathit{mapFB} :: (\mathit{elt} \to \mathit{lst} \to \mathit{lst}) \to (a \to \mathit{elt}) \to a \to \mathit{lst} \to \mathit{lst} \\
&\mathit{mapFB}\ c\ f = \lambda x\ \mathit{ys} \to c\ (f\ x)\ \mathit{ys}
\end{aligned}
$$

With that, we have all we need to convert map into build/fold.

```
{-# RULES "map" ∀ f xs. map f xs =
  build (\c n -> foldr mapFB c f) n xs) #-}
```

We need a few extra rules to convert maps into fold/builds

To convert our definition of maps into a fold/build pair, we need the following helper function.

$$mapFB :: (elt \to lst \to lst) \to (a \to elt) \to a \to lst \to lst$$
$$mapFB\ c\ f = \lambda x\ ys \to c\ (f\ x)\ ys$$

With that, we have all we need to convert map into build/fold.

```
{-# RULES "map" ∀ f xs. map f xs =
  build (\c n -> foldr mapFB c f) n xs) #-}
```

Manual rewrite rule application

Let's try applying the rewrite rules manually.

$$sum\ (map\ sq\ xs)$$

Manual rewrite rule application

Let's try applying the rewrite rules manually.

$$\begin{aligned} & sum\ (map\ sq\ xs) \\ \equiv \quad & \{\ \text{expand}\ map\ f\ xs\ \} \end{aligned}$$

Manual rewrite rule application

Let's try applying the rewrite rules manually.

$$
\begin{aligned}
& sum\ (map\ sq\ xs) \\
\equiv &\quad \{\ \text{expand } map\ f\ xs\ \} \\
& sum\ (build\ (\lambda c\ n \rightarrow foldr\ (mapFB\ c\ sq)\ n\ xs))
\end{aligned}
$$

Manual rewrite rule application

Let's try applying the rewrite rules manually.

$$
\begin{aligned}
& sum\ (map\ sq\ xs) \\
\equiv \quad & \{\ \text{expand } map\ f\ xs\ \} \\
& sum\ (build\ (\lambda c\ n \rightarrow foldr\ (mapFB\ c\ sq)\ n\ xs)) \\
\equiv \quad & \{\ \text{expand sum}\ \}
\end{aligned}
$$

Manual rewrite rule application

Let's try applying the rewrite rules manually.

$$
\begin{aligned}
& sum\ (map\ sq\ xs) \\
\equiv \quad & \{\ \text{expand } map\ f\ xs\ \} \\
& sum\ (build\ (\lambda c\ n \rightarrow foldr\ (mapFB\ c\ sq)\ n\ xs)) \\
\equiv \quad & \{\ \text{expand sum}\ \} \\
& foldr\ (+)\ 0\ (build\ (\lambda c\ n \rightarrow foldr\ (mapFB\ c\ sq)\ n\ xs))
\end{aligned}
$$

Manual rewrite rule application

Let's try applying the rewrite rules manually.

$$
\begin{aligned}
& \mathit{sum}\ (\mathit{map}\ \mathit{sq}\ \mathit{xs}) \\
\equiv \quad & \{ \text{ expand } \mathit{map}\ f\ \mathit{xs} \ \} \\
& \mathit{sum}\ (\mathit{build}\ (\lambda c\ n \rightarrow \mathit{foldr}\ (\mathit{mapFB}\ c\ \mathit{sq})\ n\ \mathit{xs})) \\
\equiv \quad & \{ \text{ expand sum } \} \\
& \mathit{foldr}\ (+)\ 0\ (\mathit{build}\ (\lambda c\ n \rightarrow \mathit{foldr}\ (\mathit{mapFB}\ c\ \mathit{sq})\ n\ \mathit{xs})) \\
\equiv \quad & \{ \text{ apply } \mathit{foldr}\ /\ \mathit{build}\text{: } \mathit{foldr}\ f\ z\ (\mathit{build}\ g) = g\ f\ z \ \}
\end{aligned}
$$

Manual rewrite rule application

Let's try applying the rewrite rules manually.

$$
\begin{aligned}
& sum\ (map\ sq\ xs) \\
\equiv \quad & \{ \text{ expand } map\ f\ xs\ \} \\
& sum\ (build\ (\lambda c\ n \rightarrow foldr\ (mapFB\ c\ sq)\ n\ xs)) \\
\equiv \quad & \{ \text{ expand sum } \} \\
& foldr\ (+)\ 0\ (build\ (\lambda c\ n \rightarrow foldr\ (mapFB\ c\ sq)\ n\ xs)) \\
\equiv \quad & \{ \text{ apply } foldr\ /\ build\text{: } foldr\ f\ z\ (build\ g) = g\ f\ z\ \} \\
& \lambda c\ n \rightarrow foldfuse\ (mapFB\ c\ sq)\ n\ xs)\ (+)\ 0
\end{aligned}
$$

Manual rewrite rule application

Let's try applying the rewrite rules manually.

$$
\begin{aligned}
& sum\ (map\ sq\ xs) \\
& \equiv \quad \{\ \text{expand } map\ f\ xs\ \} \\
& sum\ (build\ (\lambda c\ n \to foldr\ (mapFB\ c\ sq)\ n\ xs)) \\
& \equiv \quad \{\ \text{expand sum}\ \} \\
& foldr\ (+)\ 0\ (build\ (\lambda c\ n \to foldr\ (mapFB\ c\ sq)\ n\ xs)) \\
& \equiv \quad \{\ \text{apply } foldr\ /\ build\text{: } foldr\ f\ z\ (build\ g) = g\ f\ z\ \} \\
& \lambda c\ n \to foldfuse\ (mapFB\ c\ sq)\ n\ xs)\ (+)\ 0 \\
& \equiv \quad \{\ \text{apply lambda}\ \}
\end{aligned}
$$

Manual rewrite rule application

Let's try applying the rewrite rules manually.

$$
\begin{aligned}
& \mathit{sum}\ (\mathit{map}\ \mathit{sq}\ \mathit{xs}) \\
\equiv \quad & \{\ \text{expand } \mathit{map}\ f\ \mathit{xs}\ \} \\
& \mathit{sum}\ (\mathit{build}\ (\lambda c\ n \to \mathit{foldr}\ (\mathit{mapFB}\ c\ \mathit{sq})\ n\ \mathit{xs})) \\
\equiv \quad & \{\ \text{expand sum}\ \} \\
& \mathit{foldr}\ (+)\ 0\ (\mathit{build}\ (\lambda c\ n \to \mathit{foldr}\ (\mathit{mapFB}\ c\ \mathit{sq})\ n\ \mathit{xs})) \\
\equiv \quad & \{\ \text{apply } \mathit{foldr}\ /\ \mathit{build}\text{: } \mathit{foldr}\ f\ z\ (\mathit{build}\ g) = g\ f\ z\ \} \\
& \lambda c\ n \to \mathit{foldfuse}\ (\mathit{mapFB}\ c\ \mathit{sq})\ n\ \mathit{xs})\ (+)\ 0 \\
\equiv \quad & \{\ \text{apply lambda}\ \} \\
& \mathit{foldfuse}\ (\lambda x\ \mathit{ys} \to \mathit{sq}\ x + \mathit{ys})\ 0\ \mathit{xs}
\end{aligned}
$$

Applying foldr: the empty case

We now look at empty case

$$foldfuse\ (\lambda x\ ys \rightarrow sq\ x + ys)\ 0\ [\,]$$

Applying foldr: the empty case

We now look at empty case

$$foldfuse\ (\lambda x\ ys \rightarrow sq\ x + ys)\ 0\ [\,]$$

$\equiv$ {-expand $foldr$ case: $foldr\ f\ z\ [\,] =$ z -}

Applying foldr: the empty case

We now look at empty case

$$foldfuse\ (\lambda x\ ys \rightarrow sq\ x + ys)\ 0\ [\,]$$

$\equiv$ {-expand $foldr$ case: $foldr\ f\ z\ [\,] =$ z -}

$$0$$

Applying foldr: the list case

Now let's do the $(x : xs)$ case.

$$process\ (x : xs) = foldfuse\ (\lambda x\ ys \to sq\ x + ys)\ 0\ (x : xs)$$

Applying foldr: the list case

Now let's do the $(x : xs)$ case.

$$process\ (x : xs) = foldfuse\ (\lambda x\ ys \rightarrow sq\ x + ys)\ 0\ (x : xs)$$

$$\equiv\ \{\text{-expand } foldr \text{ case: } foldr\ f\ z\ (x : xs) = f\ x\ (foldr\ f\ z\ xs) \text{ -}\}$$

Applying foldr: the list case

Now let's do the $(x : xs)$ case.

$$process\ (x : xs) = foldfuse\ (\lambda x\ ys \rightarrow sq\ x + ys)\ 0\ (x : xs)$$

$\equiv$ {-expand $foldr$ case: $foldr\ f\ z\ (x : xs) = f\ x\ (foldr\ f\ z\ xs)$ -}

$$(\lambda x\ ys \rightarrow sq\ x + ys)\ x\ (foldr\ (\lambda x\ ys \rightarrow sq\ x + ys)\ z\ xs)$$

Applying foldr: the list case

Now let's do the $(x : xs)$ case.

$$process\ (x : xs) = foldfuse\ (\lambda x\ ys \rightarrow sq\ x + ys)\ 0\ (x : xs)$$

$\equiv$ {-expand $foldr$ case: $foldr\ f\ z\ (x : xs) = f\ x\ (foldr\ f\ z\ xs)$ -}

$$(\lambda x\ ys \rightarrow sq\ x + ys)\ x\ (foldr\ (\lambda x\ ys \rightarrow sq\ x + ys)\ z\ xs)$$

$\equiv$ {-use definition of $processFuse$: $foldr\ f\ 0\ xs = processFuse\ xs$ -

Applying foldr: the list case

Now let's do the $(x : xs)$ case.

$$process\ (x : xs) = foldfuse\ (\lambda x\ ys \rightarrow sq\ x + ys)\ 0\ (x : xs)$$

$$\equiv\ \{\text{-expand } foldr \text{ case: } foldr\ f\ z\ (x : xs) = f\ x\ (foldr\ f\ z\ xs) \text{ -}\}$$

$$(\lambda x\ ys \rightarrow sq\ x + ys)\ x\ (foldr\ (\lambda x\ ys \rightarrow sq\ x + ys)\ z\ xs)$$

$$\equiv\ \{\text{-use definition of } processFuse\text{: } foldr\ f\ 0\ xs = processFuse\ xs \text{ -}$$

$$(\lambda x\ ys \rightarrow sq\ x + ys)\ x\ (processFuse\ xs)$$

Applying foldr: the list case

Now let's do the $(x : xs)$ case.

$$process\ (x : xs) = foldfuse\ (\lambda x\ ys \rightarrow sq\ x + ys)\ 0\ (x : xs)$$

$\equiv$ {-expand $foldr$ case: $foldr\ f\ z\ (x : xs) = f\ x\ (foldr\ f\ z\ xs)$ -}

$$(\lambda x\ ys \rightarrow sq\ x + ys)\ x\ (foldr\ (\lambda x\ ys \rightarrow sq\ x + ys)\ z\ xs)$$

$\equiv$ {-use definition of $processFuse$: $foldr\ f\ 0\ xs = processFuse\ xs$ -

$$(\lambda x\ ys \rightarrow sq\ x + ys)\ x\ (processFuse\ xs)$$

$\equiv$ {-apply lambda -}

Applying foldr: the list case

Now let's do the $(x : xs)$ case.

$$process\ (x : xs) = foldfuse\ (\lambda x\ ys \rightarrow sq\ x + ys)\ 0\ (x : xs)$$

$$\equiv\ \{\text{-expand } foldr \text{ case: } foldr\ f\ z\ (x : xs) = f\ x\ (foldr\ f\ z\ xs)\ \text{-}\}$$

$$(\lambda x\ ys \rightarrow sq\ x + ys)\ x\ (foldr\ (\lambda x\ ys \rightarrow sq\ x + ys)\ z\ xs)$$

$$\equiv\ \{\text{-use definition of } processFuse\text{: } foldr\ f\ 0\ xs = processFuse\ xs\ \text{-}$$

$$(\lambda x\ ys \rightarrow sq\ x + ys)\ x\ (processFuse\ xs)$$

$$\equiv\ \{\text{-apply lambda -}\}$$

$$sq\ x + process\ xs$$

Applying foldr: the list case

Now let's do the $(x : xs)$ case.

$$process\ (x : xs) = foldfuse\ (\lambda x\ ys \rightarrow sq\ x + ys)\ 0\ (x : xs)$$

$\equiv$ {-expand $foldr$ case: $foldr\ f\ z\ (x : xs) = f\ x\ (foldr\ f\ z\ xs)$ -}

$$(\lambda x\ ys \rightarrow sq\ x + ys)\ x\ (foldr\ (\lambda x\ ys \rightarrow sq\ x + ys)\ z\ xs)$$

$\equiv$ {-use definition of $processFuse$: $foldr\ f\ 0\ xs = processFuse\ xs$ -

$$(\lambda x\ ys \rightarrow sq\ x + ys)\ x\ (processFuse\ xs)$$

$\equiv$ {-apply lambda -}

$$sq\ x + process\ xs$$

$\equiv$ {-inline sq -}

$$x * x + processFuse\ xs$$

Bringing both cases back together

If we now combine our two cases, we have the following

$$process_{hand}\ [\,] = 0$$
$$process_{hand}\ (x : xs) = x * x + process_{hand}\ xs$$

This is the same as what we had originally written manually!

We achieved list fusion using $foldr$ / $build$ with rewrite rules

We managed to fuse $process$ using our rewrite rules. We can look at the output of the compiler and it confirms what we expected.

$$process_{hand}\ [\,] = 0$$
$$process_{hand}\ (x : xs) = x * x + process_{hand}\ xs$$

We achieved list fusion using *foldr* / *build* with rewrite rules

We managed to fuse *process* using our rewrite rules. We can look at the output of the compiler and it confirms what we expected.

$$process_{hand}\ [\,] = 0$$
$$process_{hand}\ (x : xs) = x * x + process_{hand}\ xs$$

Function	Time (ms)	Memory (MB)
$process$	41.86	265.26
$process'$	25.31	96.65
$process_{hand}$	25.31	96.65
$processFuse$	25.31	96.65

Table of Contents

Introduction to Stream

The Stream fusion system attempts to do something similar, by defining a list as a state machine.

$$
\begin{array}{l}
\textbf{data}\ \mathit{Stream}\ a\ \textbf{where} \\
\quad \mathit{Stream} :: (s \rightarrow \mathit{Step}\ a\ s) \rightarrow s \rightarrow \mathit{Stream}\ a
\end{array}
$$

Introduction to Stream

The Stream fusion system attempts to do something similar, by defining a list as a state machine.

$$
\begin{array}{l}
\textbf{data}\ \mathit{Stream}\ a\ \textbf{where} \\
\quad \mathit{Stream} :: (s \to \mathit{Step}\ a\ s) \to s \to \mathit{Stream}\ a
\end{array}
$$

$$
\begin{array}{lcll}
\textbf{data}\ \mathit{Step}\ a\ s & = & \mathit{Done} & \\
 & \mid & \mathit{Skip} & s \\
 & \mid & \mathit{Yield}\ a & s
\end{array}
$$

Streams have little helpers to make lists

To work on standard lists, we introduce the following two functions to convert between lists and streams.

$$
\begin{aligned}
&steam &&:: [\,a\,] \rightarrow Stream\ a \\
&unstream &&:: Stream\ a \rightarrow [\,a\,]
\end{aligned}
$$

Note that these functions are inverses.

$$
\begin{aligned}
&stream \circ unstream \equiv id_{stream} \\
&unstream \circ stream \equiv id_{[a]}
\end{aligned}
$$

Maps on Streams!

$$
\begin{array}{l}
\mathit{maps} :: (a \to b) \to \mathit{Stream}\ a \to \mathit{Stream}\ b \\
\mathit{maps}\ f\ (\mathit{Stream}\ \mathit{next0}\ \mathit{s0}) = \mathit{Stream}\ \mathit{next}\ \mathit{s0} \\
\quad \mathbf{where} \\
\qquad \mathit{next}\ s = \mathbf{case}\ \mathit{next0}\ s\ \mathbf{of} \\
\qquad\quad \mathit{Done} \to \mathit{Done} \\
\qquad\quad \mathit{Skip}\ s' \to \mathit{Skip}\ s' \\
\qquad\quad \mathit{Yield}\ x\ s' \to \mathit{Yield}\ (f\ x)\ s'
\end{array}
$$

Maps on Streams!

$$
\begin{array}{l}
\mathit{maps} :: (a \to b) \to \mathit{Stream}\ a \to \mathit{Stream}\ b \\
\mathit{maps}\ f\ (\mathit{Stream}\ \mathit{next0}\ \mathit{s0}) = \mathit{Stream}\ \mathit{next}\ \mathit{s0} \\
\quad \mathbf{where} \\
\qquad \mathit{next}\ s = \mathbf{case}\ \mathit{next0}\ s\ \mathbf{of} \\
\qquad\quad \mathit{Done} \to \mathit{Done} \\
\qquad\quad \mathit{Skip}\ s' \to \mathit{Skip}\ s' \\
\qquad\quad \mathit{Yield}\ x\ s' \to \mathit{Yield}\ (f\ x)\ s'
\end{array}
$$

$$
\begin{array}{l}
\mathit{mapl} :: (a \to b) \to [a] \to [b] \\
\mathit{mapl}\ f = \mathit{unstream} \circ \mathit{maps}\ f \circ \mathit{stream}
\end{array}
$$

Stream Fusion!

Fusion on streams only has one rewrite rule, and it is pretty simple.

```
{-# RULES "stream" ∀ (s :: Stream a).
    stream (unstream s) = s #-}
```

Stream Fusion!

Fusion on streams only has one rewrite rule, and it is pretty simple.

```
{-# RULES "stream" ∀ (s :: Stream a).
    stream (unstream s) = s #-}
```

$$
\begin{aligned}
&\mathit{mapTestStream} :: [\mathit{Int}] \to [\mathit{Int}] \\
&\mathit{mapTestStream}\ \mathit{xs} = \mathit{mapl}\ (+1)\ (\mathit{mapl}\ (*2)\ \mathit{xs})
\end{aligned}
$$

Stream Fusion!

Fusion on streams only has one rewrite rule, and it is pretty simple.

```
{-# RULES "stream" ∀ (s :: Stream a).
    stream (unstream s) = s #-}
```

$$
\begin{aligned}
&\mathit{mapTestStream} :: [\mathit{Int}] \to [\mathit{Int}] \\
&\mathit{mapTestStream}\ \mathit{xs} = \mathit{mapl}\ (+1)\ (\mathit{mapl}\ (*2)\ \mathit{xs})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{mapTestStreamCompiled} :: [\mathit{Int}] \to [\mathit{Int}] \\
&\mathit{mapTestStreamCompiled}\ [\,] = [\,] \\
&\mathit{mapTestStreamCompiled}\ (x : \mathit{xs}) = \\
&\quad 1 + (x * 2) : \mathit{mapTestStreamCompiled}\ \mathit{xs}
\end{aligned}
$$

Table of Contents

We can make $process$ even faster with $Data.Vector$

The $Data.Vector$ package uses stream fusion and many other rewrite rules behind the scenes in order to optimize array based computations.

$$process\ xs = sum0 \circ map\ sq\ \$\ xs$$

We can make $process$ even faster with $Data.Vector$

The $Data.Vector$ package uses stream fusion and many other rewrite rules behind the scenes in order to optimize array based computations.

$$process\ xs = sum0 \circ map\ sq\ \$\ xs$$

The vector version looks very similar.

$$\textbf{import}\ qualified\ Data.Vector\ as\ V$$

$$processVec\ n = V.sum\ \$\ V.map\ sq\ \$\ V.enumFromTo\ 1\ (n :: Int)$$

We can make *process* even faster with *Data.Vector*

But has incredible performance!

Function	Time (ms)	Memory (MB)
process	41.86	265.26
processFuse	25.31	96.65
processVec	0.7	16×10^{-5}

What code does $Data.Vector$ generate?

While we wrote this in our program

$$processVec\ n = V.sum\ \$\ V.map\ sq\ \$\ V.enumFromTo\ 1\ (n :: Int)$$

GHC then generates the following code (simplified back to Haskell).

What code does $Data.Vector$ generate?

While we wrote this in our program

$$processVec\ n = V.sum\ \$\ V.map\ sq\ \$\ V.enumFromTo\ 1\ (n :: Int)$$

GHC then generates the following code (simplified back to Haskell).

$$\begin{aligned}
&processVecGHC\ n = loop\ 1\ 0\\
&\quad \textbf{where}\\
&\qquad loop\ count\ acc = \textbf{case}\ count \leqslant n\ \textbf{of}\\
&\qquad\quad False \rightarrow acc\\
&\qquad\quad True \rightarrow loop\ (count + 1)\ (acc + (count * count))
\end{aligned}$$

Repa: A numerical Haskell Library using Fusion

Repa also uses fusion in order to handle parallel array operations.

$$\textbf{import}\ \mathit{qualified}\ \mathit{Data.Array.Repa}\ \mathit{as}\ R$$

$$\mathit{processRepa}\ n = R.\mathit{foldP}\ (+)\ 0 \circ R.\mathit{map}\ \mathit{sq}\ \$\ \mathit{array}$$
$$\quad \textbf{where}$$
$$\qquad \mathit{array} = R.\mathit{fromListUnboxed}\ (R.Z\ R.:.\ (n :: \mathit{Int}))\ [1\mathinner{..}n]$$