Fusion: Applying Equational Transforms to Simplify Programs

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

Ryan Orendorff

May 2017

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

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. [1–4]

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

Where we have defined the functions as follows.

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. [1–4]

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

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

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. [1–4]

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

Where we have defined the functions as follows.

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

$$sq\ x = x * x$$

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

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

How fast is $process$?

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

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

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

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

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

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 $foldr$ 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 $foldr$ 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 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.

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

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

We can get good performance with manual code

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

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

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

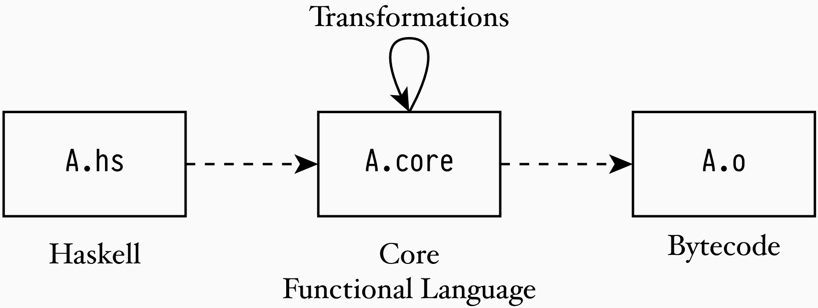
Function	Time (ms)	Memory (MB)
$process$	41.86	265.26
$process'$	25.31	96.65
$process_{hand}$	26.80	96.65

It seems we have matched GHC's performance!

Table of Contents

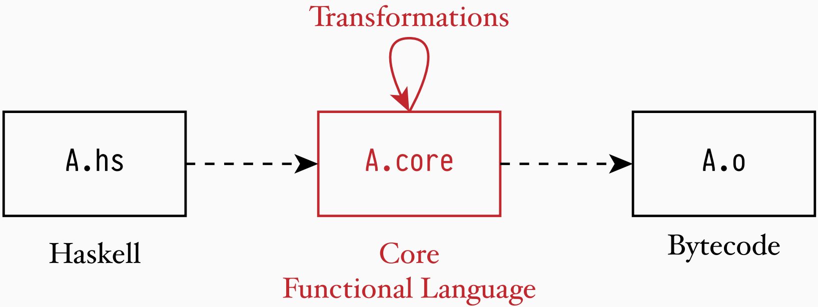
The GHC Compilation Pipeline converts Haskell into an intermediate language and then bytecode

When GHC compiles a Haskell program, it converts the code into an intermediate language called "Core", which is then (eventually) turned into byte code. [5]



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. [5]



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. [6]

- 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. [6]

- Inlining functions
- Applying a function to its arguments
  $((\lambda x \rightarrow x + y)\, 4 \Rightarrow 4 + y)$

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. [6]

- Inlining functions
- Applying a function to its arguments
  $((\lambda x \rightarrow x + y)\, 4 \Rightarrow 4 + y)$
- Simplifying constant expressions $(x + 8 - 1 \Rightarrow x + 7)$

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. [6]

- Inlining functions
- Applying a function to its arguments
  $((\lambda x \rightarrow x + y)\, 4 \Rightarrow 4 + y)$
- Simplifying constant expressions $(x + 8 - 1 \Rightarrow x + 7)$
- 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. [6]

- Inlining functions
- Applying a function to its arguments
  $((\lambda x \rightarrow x + y)\ 4 \Rightarrow 4 + y)$
- Simplifying constant expressions $(x + 8 - 1 \Rightarrow x + 7)$
- 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. [6]

- Inlining functions
- Applying a function to its arguments $((\lambda x \to x + y)\ 4 \Rightarrow 4 + y)$
- Simplifying constant expressions $(x + 8 - 1 \Rightarrow x + 7)$
- 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. [7]

```
{-# 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. [7]

```
{-# 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. [7]

```
{-# 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). This annotation is optional.

Rewrite Rules allow us to say two expressions are equivalent

Rewrite rules allow us to replace terms in the program with equivalent terms. [7]

```
{-# 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). This annotation is optional.
- The `forall` brings a variable into scope. Sometimes written $\forall$.

Rewrite Rules allow us to say two expressions are equivalent

Rewrite rules allow us to replace terms in the program with equivalent terms. [7]

```
{-# 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). This annotation is optional.
- The `forall` brings a variable into scope. Sometimes written $\forall$.
- After the period are the equivalent statements.

Rules have some restrictions

Rewrite rules have some gotchas. [8]

- Rules don't prevent you from doing something silly

```
{-# RULES "id5" forall (x :: Int). id x = 5 #-}
```

Rules have some restrictions

Rewrite rules have some gotchas. [8]

- Rules don't prevent you from doing something silly

  ```
  {-# RULES "id5" forall (x :: Int). 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 #-}
  ```

  $x \nRightarrow id\ x$

Rules have some restrictions

Rewrite rules have some gotchas. [8]

- Rules don't prevent you from doing something silly

  ```
  {-# RULES "id5" forall (x :: Int). 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 #-}
  ```

  $x \nRightarrow id\ x$

- 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. [8]

- Rules don't prevent you from doing something silly

  ```
  {-# RULES "id5" forall (x :: Int). 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 #-}
  ```

  $x \nRightarrow id\ x$

- 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 arbitrarily chooses one.

We can combine maps to traverse a list once

Let us introduce the following rule about maps. [4]

```
{-# 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. [4]

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

$$mapTest :: [Int] \to [Int]$$
$$mapTest\ xs = map\ (+1)\ (map\ (*2)\ xs)$$

$$mapTest_{fuse} :: [Int] \to [Int]$$
$$mapTest_{fuse}\ xs = map_{fuse}\ (+1)\ (map_{fuse}\ (*2)\ xs)$$

Our map fusion performs (a bit) better!

We can test our functions on a million elements

$$mapTest\ xs = map\ (+1)\ (map\ (*2)\ xs)$$
$$mapTest_{fuse}\ xs = map_{fuse}\ (+1)\ (map_{fuse}\ (*2)\ xs)$$

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

Rules allow us to perform *fusion*, where we remove intermediate data structures from the computation.

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

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

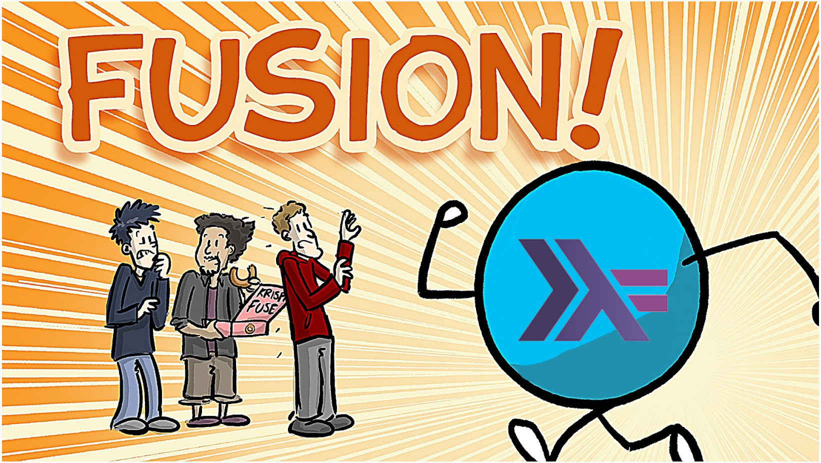
$$process\ xs = sum \circ map\ sq\ \$\ xs$$

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

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

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

FUSION!



From PhD Comics

Table of Contents

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

GHC accomplishes fusion with two functions: foldr and build. [3, 9]

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

GHC accomplishes fusion with two functions: foldr and build. [3, 9]

$foldr$ combines the elements of a list

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

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

GHC accomplishes fusion with two functions: foldr and build. [3, 9]

$foldr$ 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.

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

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

GHC accomplishes fusion with two functions: foldr and build. [3, 9]

foldr combines the elements of a list

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

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

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

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

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 #-}
```

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

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. [8, 9]

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

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. [8, 9]

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

As an example, lets apply the list cons $c = (:)$ and $f = sq$

$$\lambda x\ ys \to sq\ x : ys$$

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

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

```
{-# RULES "map" [~1] ∀ 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

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

```
{-# RULES "map" [~1] ∀ f xs. map f xs =
  build (\c n -> foldr (mapFB c f) n xs) #-}
```

We also provide a way to cancel failed fusion by converting back to a map.

```
{-# RULES "mapList" [1] ∀ f.
  foldr (mapFB (:) f) []  = map f #-}
```

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

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

```
{-# RULES "map" [~1] ∀ f xs. map f xs =
  build (\c n -> foldr (mapFB c f) n xs) #-}
```

We also provide a way to cancel failed fusion by converting back to a map.

```
{-# RULES "mapList" [1] ∀ f.
  foldr (mapFB (:) f) []  = map f #-}
```

$$
\begin{aligned}
& build\ (\lambda c\ n \rightarrow foldr\ (mapFB\ c\ f)\ n\ xs) \\
\equiv\ & \{\text{-inline def of build -}\} \\
& (\lambda c\ n \rightarrow foldr\ (mapFB\ c\ f)\ n\ xs)\ (:)\ [\,] \\
\equiv\ & \{\text{-remove lambda -}\} \\
& foldr\ (mapFB\ (:)\ f)\ [\,]\ xs
\end{aligned}
$$

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.

$$
\begin{array}{ll}
\multicolumn{2}{l}{sum\ (map\ sq\ xs)} \\
\equiv & \{\ \text{expand } map\ f\ xs\ \}
\end{array}
$$

Manual rewrite rule application

Let's try applying the rewrite rules manually.

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

Manual rewrite rule application

Let's try applying the rewrite rules manually.

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

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.

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

Manual rewrite rule application

Let's try applying the rewrite rules manually.

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

Manual rewrite rule application

Let's try applying the rewrite rules manually.

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

Manual rewrite rule application

Let's try applying the rewrite rules manually.

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

Applying foldr: the empty case

We now look at empty case

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

Applying foldr: the empty case

We now look at empty case

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

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

Applying foldr: the empty case

We now look at empty case

$$foldr\ (\lambda x\ ys \to sq\ x + ys)\ 0\ [\,]$$

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

$$0$$

Applying foldr: the list case

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

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

Applying foldr: the list case

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

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

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

Applying foldr: the list case

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

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

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

$$(\lambda x\ ys \rightarrow sq\ x + ys)\ x\ (foldr\ (\lambda x\ ys \rightarrow sq\ x + ys)\ 0\ xs)$$

Applying foldr: the list case

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

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

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

$$(\lambda x\ ys \rightarrow sq\ x + ys)\ x\ (foldr\ (\lambda x\ ys \rightarrow sq\ x + ys)\ 0\ xs)$$

$$\equiv\ \{\text{-use def of } process_{fuse}\text{: } foldr\ f\ 0\ xs = process_{fuse}\ xs \text{ -}\}$$

Applying foldr: the list case

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

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

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

$$(\lambda x\ ys \rightarrow sq\ x + ys)\ x\ (foldr\ (\lambda x\ ys \rightarrow sq\ x + ys)\ 0\ xs)$$

$$\equiv\ \{\text{-use def of } process_{fuse}\text{: } foldr\ f\ 0\ xs = process_{fuse}\ xs \text{ -}\}$$

$$(\lambda x\ ys \rightarrow sq\ x + ys)\ x\ (process_{fuse}\ xs)$$

Applying foldr: the list case

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

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

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

$$(\lambda x\ ys \to sq\ x + ys)\ x\ (foldr\ (\lambda x\ ys \to sq\ x + ys)\ 0\ xs)$$

$\equiv$ {-use def of $process_{fuse}$: $foldr\ f\ 0\ xs = process_{fuse}\ xs$ -}

$$(\lambda x\ ys \to sq\ x + ys)\ x\ (process_{fuse}\ xs)$$

$\equiv$ {-apply lambda -}

Applying foldr: the list case

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

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

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

$$(\lambda x\ ys \rightarrow sq\ x + ys)\ x\ (foldr\ (\lambda x\ ys \rightarrow sq\ x + ys)\ 0\ xs)$$

$$\equiv\ \{\text{-use def of } process_{fuse}\text{: } foldr\ f\ 0\ xs = process_{fuse}\ xs\ \text{-}\}$$

$$(\lambda x\ ys \rightarrow sq\ x + ys)\ x\ (process_{fuse}\ xs)$$

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

$$sq\ x + process\ xs$$

Applying foldr: the list case

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

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

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

$$(\lambda x\ ys \rightarrow sq\ x + ys)\ x\ (foldr\ (\lambda x\ ys \rightarrow sq\ x + ys)\ 0\ xs)$$

$$\equiv\ \{\text{-use def of } process_{fuse}\text{: } foldr\ f\ 0\ xs = process_{fuse}\ xs\ \text{-}\}$$

$$(\lambda x\ ys \rightarrow sq\ x + ys)\ x\ (process_{fuse}\ xs)$$

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

$$sq\ x + process\ xs$$

$$\equiv\ \{\text{-inline } sq \text{ -}\}$$

$$x * x + process_{fuse}\ xs$$

Bringing both cases back together

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

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

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

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

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.

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

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.

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

As expected, we get the same performance after performing the fusion rules.

Function	Time (ms)	Memory (MB)
$process$	41.86	265.26
$process'$	26.60	96.65
$process_{hand}$	28.80	96.65
$process_{fuse}$	27.08	96.65

There are many types of fusion concepts out there

While $foldr$ / $build$ works well, it can have problems fusing zip and $foldl$.

There are a few other systems out there. [2, 3]

- $unbuild$ / $unfoldr$, where $unfoldr$ builds a list and $unbuild$ consumes a list. It can have problems fusing $filter$.

There are many types of fusion concepts out there

While $foldr$ / $build$ works well, it can have problems fusing zip and $foldl$.

There are a few other systems out there. [2, 3]

- $unbuild$ / $unfoldr$, where $unfoldr$ builds a list and $unbuild$ consumes a list. It can have problems fusing $filter$.
- stream fusion, which works by defining a $Stream$ data type that acts like an iterator.

Table of Contents

Introduction to Stream

The Stream fusion system attempts to do something similar, by defining a list as an iterator. [2, 3]

$$\begin{aligned}&\mathbf{data}\ \mathit{Stream}\ a\ \mathbf{where}\\&\quad \mathit{Stream} :: (s \to \mathit{Step}\ a\ s) \to s \to \mathit{Stream}\ a\end{aligned}$$

Introduction to Stream

The Stream fusion system attempts to do something similar, by defining a list as an iterator. [2, 3]

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Stream}\ a\ \mathbf{where} \\
&\quad \mathit{Stream} :: (s \to \mathit{Step}\ a\ s) \to s \to \mathit{Stream}\ a
\end{aligned}
$$

where $\mathit{Step}\ a\ s$ informs us how to keep processing the stream.

$$
\begin{aligned}
\mathbf{data}\ \mathit{Step}\ a\ s = {} & \mathit{Done} \\
\mid {} & \mathit{Skip}\ \ \ \ \ \ s \\
\mid {} & \mathit{Yield}\ a\ s
\end{aligned}
$$

Streams have little helpers to make lists: stream

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

$$
\begin{aligned}
&\mathit{stream} :: [\,a\,] \to \mathit{Stream}\ a \\
&\mathit{stream}\ \mathit{xs} = \mathit{Stream}\ \mathit{uncons}\ \mathit{xs} \\
&\quad \mathbf{where} \\
&\qquad \mathit{uncons}\ [\,] \qquad\quad = \mathit{Done} \\
&\qquad \mathit{uncons}\ (y : \mathit{ys}) = \mathit{Yield}\ y\ \mathit{ys}
\end{aligned}
$$

Streams have little helpers to make lists: unstream

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

$$
\begin{array}{l}
\mathit{unstream} :: \mathit{Stream}\ a \rightarrow [\,a\,] \\
\mathit{unstream}\ (\mathit{Stream}\ \mathit{next}\ \mathit{s0}) = \mathit{unfold}\ \mathit{next}\ \mathit{s0} \\
\quad \mathbf{where} \\
\qquad \mathit{unfold}\ \mathit{next}\ s = \mathbf{case}\ \mathit{next}\ s\ \mathbf{of} \\
\qquad\quad \begin{array}{lll}
\mathit{Done} & \rightarrow [\,] & \\
\mathit{Skip}\ s' & \rightarrow & \mathit{unfold}\ \mathit{next}\ s' \\
\mathit{Yield}\ x\ s' & \rightarrow x : & \mathit{unfold}\ \mathit{next}\ s'
\end{array}
\end{array}
$$

Let's define map for $Streams$

We can define some standard list processing functions on $Streams$. Let's try map.

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

Let's define map for $Streams$

We can define some standard list processing functions on $Streams$. Let's try map.

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

$$
\begin{aligned}
& map_{[a]} :: (a \to b) \to [\,a\,] \to [\,b\,] \\
& map_{[a]}\ f = unstream \circ map_s\ f \circ stream
\end{aligned}
$$

Fusion on Streams

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

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

Fusion on Streams

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

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

$$mapTestStream :: [Int] \rightarrow [Int]$$
$$mapTestStream\ xs = map_{[a]}\ (+1) \circ map_{[a]}\ (*2)\ \$\ xs$$

Fusion on Streams

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

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

$$mapTestStream :: [Int] \rightarrow [Int]$$
$$mapTestStream\ xs = map_{[a]}\ (+1) \circ map_{[a]}\ (*2)\ \$\ xs$$

$$map_{[a]}\ (+1) \circ map_{[a]}\ (*2)$$
$$\equiv\ \{\text{-expand mapl -}\}$$
$$unstream \circ map_s\ (+1) \circ stream \circ unstream \circ map_s\ (*2) \circ stream$$
$$\equiv\ \{\text{-apply ''stream/unstream'' -}\}$$
$$unstream \circ map_s\ (+1) \circ map_s\ (*2) \circ stream$$

Map fused by Stream Fusion

Our map example

$$
\begin{aligned}
&mapTestStream :: [Int] \rightarrow [Int] \\
&mapTestStream\ xs = map_{[a]}\ (+1) \circ map_{[a]}\ (*2)\ \$\ xs
\end{aligned}
$$

gets fused into this result.

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

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.

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

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.

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

The vector version looks very similar.

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

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

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

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

But has awesome performance!

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

What code does $Data.Vector$ generate?

The $processVec$ function is pretty simple in Haskell itself.

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

When compiling, GHC fires *202 rules!*

What code does $Data.Vector$ generate?

The $processVec$ function is pretty simple in Haskell itself.

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

When compiling, GHC fires *202 rules!*

Specifically, this appears when using the debug flag `-ddump-rule-firings`. [10]

```
...
Rule fired: stream/unstream [Vector]
Rule fired: stream/unstream [Vector]
...
```

What code does $Data.Vector$ generate?

The $processVec$ function is pretty simple in Haskell itself.

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

When compiling, GHC fires *202 rules!*

Specifically, this appears when using the debug flag `-ddump-rule-firings`. [10]

```
...
Rule fired: stream/unstream [Vector]
Rule fired: stream/unstream [Vector]
...
```

What code does $Data.Vector$ generate?

The $processVec$ function is pretty simple in Haskell itself.

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

And the final code generated is the following.

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

Other use cases for fusion

Besides vector, fusion is used in a few other places.

- Repa, a parallel list processing library [11]

Other use cases for fusion

Besides vector, fusion is used in a few other places.

- Repa, a parallel list processing library [11]
- Vector instructions by SIMD [12]

Other use cases for fusion

Besides vector, fusion is used in a few other places.

- Repa, a parallel list processing library [11]
- Vector instructions by SIMD [12]
- Pipes, a stream processing library [13]

Wrap up

What did we talk about today?

- Goal: simple code that performed as well as a optimized version.
- A brief introduction to compilation in GHC and rewrite rules.
- $foldr$ / $build$ fusion.
- Showed a second type of fusion: stream fusion.
- Went through some libraries using fusion.

Thanks!

References I

[1] P. Wadler, "Deforestation: Transforming programs to eliminate trees," *Theoretical computer science*, vol. 73, no. 2, pp. 231–248, 1990.

[2] D. Coutts, R. Leshchinskiy, and D. Stewart, "Stream fusion: From lists to streams to nothing at all," *ACM SIGPLAN Notices*, 2007.

[3] D. Coutts, "Stream Fusion," Ph.D. dissertation, University of Oxford, Oct. 2010.

[4] M. Karpov. (2016, Nov.) GHC Optimization and Fusion. [Online]. Available: https://www.stackbuilders.com/tutorials/haskell/ghc-optimization-and-fusion/

References II

[5] G. Team. Compiling one module: HscMain. [Online]. Available: https://ghc.haskell.org/trac/ghc/wiki/Commentary/Compiler/HscMain

[6] ——. Core-to-Core optimization pipeline. [Online]. Available: https://ghc.haskell.org/trac/ghc/wiki/Commentary/Compiler/Core2CorePipeline

[7] (2010, Jan.) Playing by the Rules. [Online]. Available: https://wiki.haskell.org/Playing_by_the_rules#Fast_ByteString_construction

[8] G. Team. Using Rules. [Online]. Available: wiki.haskell.org/GHC/Using_rules

References III

[9] ——. GHC.Base. [Online]. Available: https://hackage.haskell.org/package/base-4.9.1.0/docs/src/GHC.Base.html#foldr

[10] R. Leshchinskiy. Vector. [Online]. Available: https://github.com/haskell/vector/blob/master/Data/Vector/Generic.hs#L2025

[11] B. Lippmeier, M. M. T. Chakravarty, G. Keller, and A. Robinson, *Data flow fusion with series expressions in Haskell*. ACM, Jan. 2014, vol. 48.

[12] G. Mainland, R. Leshchinskiy, and S. Peyton Jones, "Exploiting vector instructions with generalized stream fusion," in *the 18th ACM SIGPLAN international conference*. New York, New York, USA: ACM Press, 2013, pp. 37–12.

References IV

[13] G. Gonzalez. (2014, Jan.) Shortcut fusion for pipes. [Online]. Available: http://www.haskellforall.com/2014/01/stream-fusion-for-pipes.html