

Fusion: Applying Equational Transforms to Simplify Programs

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

Ryan Orendorff¹

May 2017

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

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]

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

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]

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

Where we have defined the functions as follows.

$$\text{map } [] = []$$
$$\text{map } f (x : xs) = f x : \text{map } f xs$$
$$sq x = x * x$$

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]

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

Where we have defined the functions as follows.

$$map\ [] = []$$
$$map\ f\ (x : xs) = f\ x : map\ f\ xs$$
$$sq\ x = x * x$$
$$foldr :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$$
$$foldr\ _\ z\ [] = z$$
$$foldr\ f\ z\ (x : xs) = f\ x\ (foldr\ f\ z\ xs)$$
$$sum = foldr\ (+)\ 0$$

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 *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?

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

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

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

Function	Time (ms)	Memory (MB)
<i>process</i>	41.86	265.26
<i>process'</i>	25.31	96.65

How fast is *process*?

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

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

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

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

Function	Time (ms)	Memory (MB)
<i>process</i>	41.86	265.26
<i>process'</i>	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.

$$\textit{process} :: [Int] \rightarrow Int$$
$$\textit{process} \, xs = \textit{sum} \circ \textit{map} \, \textit{sq} \, \$ \, xs$$

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.

$$process :: [Int] \rightarrow Int$$
$$process\ xs = sum \circ map\ sq\ \$\ xs$$
$$process_{hand} :: [Int] \rightarrow Int$$
$$process_{hand}\ [] = 0$$
$$process_{hand}\ (x : xs) = x * x + process_{hand}\ xs$$

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

Motivation: Simple Programs versus Performance

A brief introduction to GHC

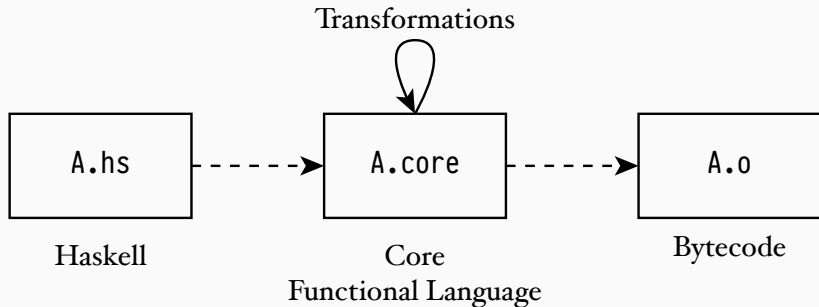
List fusion with *foldr/build*

Stream Fusion

Applications of Fusion

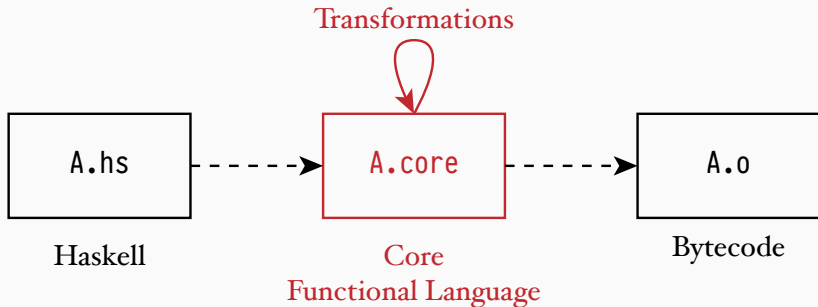
The GHC Compilation Pipeline converts Haskell into an intermediate language and then bytecode

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
 $((\lambda x \rightarrow x + y) 4 \Rightarrow 4 + y)$
- 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 doesn't prevent you from doing something silly

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

Rules have some restrictions

Rewrite rules have some gotchas. [8]

- 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 #-}
```

$x \not\Rightarrow id\ x$

Rules have some restrictions

Rewrite rules have some gotchas. [8]

- 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 #-}
```

$x \not\Rightarrow id\ x$

- 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 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 #-}
```

$x \not\Rightarrow id\ x$

- 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 #-}
```

$$\text{mapTest} :: [Int] \rightarrow [Int]$$
$$\text{mapTest } xs = \text{map } (+1) (\text{map } (*2) xs)$$
$$\text{mapTest}_{\text{fuse}} :: [Int] \rightarrow [Int]$$
$$\text{mapTest}_{\text{fuse}} xs = \text{map}_{\text{fuse}} (+1) (\text{map}_{\text{fuse}} (*2) xs)$$

Our map fusion performs (a bit) better!

We can test our functions on a million elements

$$\text{mapTest } xs = \text{map } (+1) (\text{map } (*2) xs)$$
$$\text{mapTest}_{\text{fuse}} xs = \text{map}_{\text{fuse}} (+1) (\text{map}_{\text{fuse}} (*2) xs)$$

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

Function	Time (ms)	Memory (MB)
mapTest	26.4	256.00
$\text{mapTest}_{\text{fuse}}$	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

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

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

Motivation: Simple Programs versus Performance

A brief introduction to GHC

List fusion with *foldr/build*

Stream Fusion

Applications of Fusion

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

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

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

$$\textit{foldr} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$$

$$\textit{foldr} f z [] = z$$

$$\textit{foldr} f z (x : xs) = f x (\textit{foldr} f z xs)$$

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

$$\textit{build} :: (\forall b. (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow b) \rightarrow [a]$$

$$\textit{build} g = g (:) []$$

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

$$\textit{foldr} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$$

$$\textit{foldr} f z [] = z$$

$$\textit{foldr} f z (x : xs) = f x (\textit{foldr} f z xs)$$

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

$$\textit{build} :: (\forall b. (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow b) \rightarrow [a]$$

$$\textit{build} g = g (:) []$$

$$\textit{build1} l \equiv [1, 2, 3]$$

where

$$l \text{ cons } nil = 1 \text{ 'cons' } (2 \text{ 'cons' } (3 \text{ 'cons' } nil))$$

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"
 $\forall f z (g :: \forall b. (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow b).$ 
foldr f z (build g) = g f z #-}
```

$$\text{foldr } (+) 0 (\text{build } l) \equiv l (+) 0 \equiv 1 + (2 + (3 + 0))$$

where

$$l \text{ cons } nil = 1 \text{ 'cons' } (2 \text{ 'cons' } (3 \text{ 'cons' } nil))$$

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]

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

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. [8, 9]

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

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

$$\lambda x \ \text{ys} \rightarrow sq \ x : \text{ys}$$

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]  $\forall$  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]  $\forall$  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 #-}
```

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

Manual rewrite rule application

Let's try applying the rewrite rules manually.

$$\textit{sum} (\textit{map} \textit{sq} \textit{xs})$$

Manual rewrite rule application

Let's try applying the rewrite rules manually.

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

Manual rewrite rule application

Let's try applying the rewrite rules manually.

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

Manual rewrite rule application

Let's try applying the rewrite rules manually.

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

Manual rewrite rule application

Let's try applying the rewrite rules manually.

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

Manual rewrite rule application

Let's try applying the rewrite rules manually.

sum (map sq xs)

$\equiv \{ \text{expand } \text{map } f \text{ } xs \}$

sum (build ($\lambda c \ n \rightarrow \text{foldr } (\text{mapFB } c \text{ } sq) \ n \ xs$))

$\equiv \{ \text{expand sum } \}$

foldr (+) 0 (build ($\lambda c \ n \rightarrow \text{foldr } (\text{mapFB } c \text{ } sq) \ n \ xs$))

$\equiv \{ \text{apply foldr / build: foldr } f \ z \ (\text{build } g) = g \ f \ z \}$

Manual rewrite rule application

Let's try applying the rewrite rules manually.

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

Manual rewrite rule application

Let's try applying the rewrite rules manually.

$sum\ (map\ sq\ xs)$

$\equiv \{ \text{expand } map\ f\ xs \}$

$sum\ (build\ (\lambda c\ n \rightarrow foldr\ (mapFB\ c\ sq)\ n\ xs))$

$\equiv \{ \text{expand } sum \}$

$foldr\ (+)\ 0\ (build\ (\lambda c\ n \rightarrow foldr\ (mapFB\ c\ sq)\ n\ xs))$

$\equiv \{ \text{apply } foldr\ /\ build: foldr\ f\ z\ (build\ g) = g\ f\ z \}$

$\lambda c\ n \rightarrow foldr\ (mapFB\ c\ sq)\ n\ xs\ (+)\ 0$

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

Manual rewrite rule application

Let's try applying the rewrite rules manually.

$sum\ (map\ sq\ xs)$

$\equiv \{ \text{expand } map\ f\ xs \}$

$sum\ (build\ (\lambda c\ n \rightarrow foldr\ (mapFB\ c\ sq)\ n\ xs))$

$\equiv \{ \text{expand } sum \}$

$foldr\ (+)\ 0\ (build\ (\lambda c\ n \rightarrow foldr\ (mapFB\ c\ sq)\ n\ xs))$

$\equiv \{ \text{apply } foldr / build: foldr\ f\ z\ (build\ g) = g\ f\ z \}$

$\lambda c\ n \rightarrow foldr\ (mapFB\ c\ sq)\ n\ xs\ (+)\ 0$

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

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

Applying foldr: the empty case

We now look at empty case

$$\text{foldr } (\lambda x \text{ } ys \rightarrow sq \text{ } x + ys) \text{ } 0 \text{ } []$$

Applying foldr: the empty case

We now look at empty case

$$\textit{foldr} (\lambda x \ ys \rightarrow sq \ x + ys) \ 0 \ []$$

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

Applying foldr: the empty case

We now look at empty case

$$\text{foldr } (\lambda x \text{ } ys \rightarrow sq \text{ } x + ys) \text{ } 0 \text{ } []$$

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

$$0$$

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.

$$\begin{aligned} process\ (x : xs) &= foldr\ (\lambda x\ ys \rightarrow sq\ x + ys)\ 0\ (x : xs) \\ &\equiv \{-expand\ foldr\ case:\ foldr\ f\ z\ (x : xs) = f\ x\ (foldr\ f\ z\ xs)\ -\} \end{aligned}$$

Applying foldr: the list case

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

$$\begin{aligned} process\ (x : xs) &= foldr\ (\lambda x\ ys \rightarrow sq\ x + ys)\ 0\ (x : xs) \\ &\equiv \{-expand\ foldr\ case:\ foldr\ f\ z\ (x : xs) = f\ x\ (foldr\ f\ z\ xs)\ -\} \\ &\quad (\lambda x\ ys \rightarrow sq\ x + ys)\ x\ (foldr\ (\lambda x\ ys \rightarrow sq\ x + ys)\ z\ xs) \end{aligned}$$

Applying foldr: the list case

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

$$\begin{aligned} process\ (x : xs) &= foldr\ (\lambda x\ ys \rightarrow sq\ x + ys)\ 0\ (x : xs) \\ &\equiv \{-expand\ foldr\ case:\ foldr\ f\ z\ (x : xs) = f\ x\ (foldr\ f\ z\ xs)\ -\} \\ &\quad (\lambda x\ ys \rightarrow sq\ x + ys)\ x\ (foldr\ (\lambda x\ ys \rightarrow sq\ x + ys)\ z\ xs) \\ &\equiv \{-use\ def\ of\ process_{fuse}:\ foldr\ f\ 0\ xs = process_{fuse}\ xs\ -\} \end{aligned}$$

Applying foldr: the list case

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

$$\begin{aligned} process\ (x : xs) &= foldr\ (\lambda x\ ys \rightarrow sq\ x + ys)\ 0\ (x : xs) \\ &\equiv \{-expand\ foldr\ case:\ foldr\ f\ z\ (x : xs) = f\ x\ (foldr\ f\ z\ xs)\ -\} \\ &\quad (\lambda x\ ys \rightarrow sq\ x + ys)\ x\ (foldr\ (\lambda x\ ys \rightarrow sq\ x + ys)\ z\ xs) \\ &\equiv \{-use\ def\ of\ process_{fuse}:\ foldr\ f\ 0\ xs = process_{fuse}\ xs\ -\} \\ &\quad (\lambda x\ ys \rightarrow sq\ x + ys)\ x\ (process_{fuse}\ xs) \end{aligned}$$

Applying foldr: the list case

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

$$\begin{aligned} process\ (x : xs) &= foldr\ (\lambda x\ ys \rightarrow sq\ x + ys)\ 0\ (x : xs) \\ &\equiv \{-expand\ foldr\ case:\ foldr\ f\ z\ (x : xs) = f\ x\ (foldr\ f\ z\ xs)\ -\} \\ &\quad (\lambda x\ ys \rightarrow sq\ x + ys)\ x\ (foldr\ (\lambda x\ ys \rightarrow sq\ x + ys)\ z\ xs) \\ &\equiv \{-use\ def\ of\ process_{fuse}:\ foldr\ f\ 0\ xs = process_{fuse}\ xs\ -\} \\ &\quad (\lambda x\ ys \rightarrow sq\ x + ys)\ x\ (process_{fuse}\ xs) \\ &\equiv \{-apply\ lambda\ -\} \end{aligned}$$

Applying foldr: the list case

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

$$\begin{aligned} process\ (x : xs) &= foldr\ (\lambda x\ ys \rightarrow sq\ x + ys)\ 0\ (x : xs) \\ &\equiv \{-expand\ foldr\ case:\ foldr\ f\ z\ (x : xs) = f\ x\ (foldr\ f\ z\ xs)\ -\} \\ &\quad (\lambda x\ ys \rightarrow sq\ x + ys)\ x\ (foldr\ (\lambda x\ ys \rightarrow sq\ x + ys)\ z\ xs) \\ &\equiv \{-use\ def\ of\ process_{fuse}:\ foldr\ f\ 0\ xs = process_{fuse}\ xs\ -\} \\ &\quad (\lambda x\ ys \rightarrow sq\ x + ys)\ x\ (process_{fuse}\ xs) \\ &\equiv \{-apply\ lambda\ -\} \\ &\quad sq\ x + process\ xs \end{aligned}$$

Applying foldr: the list case

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

$$\begin{aligned} process\ (x : xs) &= foldr\ (\lambda x\ ys \rightarrow sq\ x + ys)\ 0\ (x : xs) \\ &\equiv \{-expand\ foldr\ case:\ foldr\ f\ z\ (x : xs) = f\ x\ (foldr\ f\ z\ xs)\ -\} \\ &\quad (\lambda x\ ys \rightarrow sq\ x + ys)\ x\ (foldr\ (\lambda x\ ys \rightarrow sq\ x + ys)\ z\ xs) \\ &\equiv \{-use\ def\ of\ process_{fuse}:\ foldr\ f\ 0\ xs = process_{fuse}\ xs\ -\} \\ &\quad (\lambda x\ ys \rightarrow sq\ x + ys)\ x\ (process_{fuse}\ xs) \\ &\equiv \{-apply\ lambda\ -\} \\ &\quad sq\ x + process\ xs \\ &\equiv \{-inline\ sq\ -\} \\ &\quad x * x + process_{fuse}\ xs \end{aligned}$$

Bringing both cases back together

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

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

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

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

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)
<i>process</i>	41.86	265.26
<i>process'</i>	26.60	96.65
<i>process_{hand}</i>	28.80	96.65
<i>process_{fuse}</i>	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

Motivation: Simple Programs versus Performance

A brief introduction to GHC

List fusion with *foldr/build*

Stream Fusion

Applications of Fusion

Introduction to Stream

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

data *Stream a* **where**

Stream :: $(s \rightarrow \text{Step } a \ s) \rightarrow s \rightarrow \text{Stream } a$

Introduction to Stream

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

data *Stream a where*

Stream :: $(s \rightarrow \text{Step } a \ s) \rightarrow s \rightarrow \text{Stream } a$

where *Step a s* informs us how to keep processing the stream.

data *Step a s = Done*

| *Skip* *s*

| *Yield a s*

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.

$$\text{stream} :: [a] \rightarrow \text{Stream } a$$
$$\text{stream } xs = \text{Stream } \text{uncons } xs$$

where

$$\text{uncons } [] = \text{Done}$$
$$\text{uncons } (y : ys) = \text{Yield } y \text{ } ys$$

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.

$$\text{unstream} :: \text{Stream } a \rightarrow [a]$$
$$\text{unstream } (\text{Stream next } s0) = \text{unfold next } s0$$

where

$$\text{unfold next } s = \mathbf{case\ next\ } s \mathbf{\ of}$$
$$\text{Done} \quad \rightarrow []$$
$$\text{Skip } s' \quad \rightarrow \text{unfold next } s'$$
$$\text{Yield } x\ s' \rightarrow x : \text{unfold next } s'$$

Let's define *map* for *Streams*

We can define some standard list processing functions on *Streams*.

Let's try *map*.

$$\text{map}_s :: (a \rightarrow b) \rightarrow \text{Stream } a \rightarrow \text{Stream } b$$
$$\text{map}_s f (\text{Stream next0 } s0) = \text{Stream next } s0$$

where

$$\text{next } s = \mathbf{case} \text{ next0 } s \mathbf{ of}$$
$$\text{Done} \quad \rightarrow \text{Done}$$
$$\text{Skip } s' \quad \rightarrow \text{Skip} \quad s'$$
$$\text{Yield } x \ s' \rightarrow \text{Yield } (f \ x) \ s'$$

Let's define *map* for *Streams*

We can define some standard list processing functions on *Streams*.

Let's try *map*.

$$\text{map}_s :: (a \rightarrow b) \rightarrow \text{Stream } a \rightarrow \text{Stream } b$$
$$\text{map}_s f (\text{Stream next0 } s0) = \text{Stream next } s0$$

where

$$\text{next } s = \mathbf{case} \text{ next0 } s \mathbf{ of}$$
$$\text{Done} \quad \rightarrow \text{Done}$$
$$\text{Skip } s' \quad \rightarrow \text{Skip} \quad s'$$
$$\text{Yield } x \ s' \rightarrow \text{Yield } (f \ x) \ s'$$
$$\text{map}_{[a]} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$$
$$\text{map}_{[a]} f = \text{unstream} \circ \text{map}_s f \circ \text{stream}$$

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 #-}
```

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

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 #-}
```

$\text{mapTestStream} :: [Int] \rightarrow [Int]$

$\text{mapTestStream } xs = \text{map}_{[a]} (+1) \circ \text{map}_{[a]} (*2) \$ xs$

$\text{map}_{[a]} (+1) \circ \text{map}_{[a]} (*2)$

$\equiv \{-\text{expand mapl}-\}$

$\text{unstream} \circ \text{map}_s (+1) \circ \text{stream} \circ \text{unstream} \circ \text{map}_s (*2) \circ \text{stream}$

$\equiv \{-\text{apply "stream/unstream"}-\}$

$\text{unstream} \circ \text{map}_s (+1) \circ \text{map}_s (*2) \circ \text{stream}$

Our map example

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

gets fused into this result.

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

Table of Contents

Motivation: Simple Programs versus Performance

A brief introduction to GHC

List fusion with *foldr/build*

Stream Fusion

Applications of Fusion

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.

$$\textit{process} \textit{xs} = \textit{sum0} \circ \textit{map} \textit{sq} \$ \textit{xs}$$

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.

```
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 $ V.map sq $ V.enumFromTo 1 (n :: Int)
```

But has awesome performance!

Function	Time (ms)	Memory (MB)
<i>process</i>	41.86	265.26
<i>process_fuse</i>	25.31	96.65
<i>processVec</i>	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.

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

And the final code generated is the following.

```
processVecGHC n = loop 1 0
  where
    loop count acc = case count ≤ n of
      False → acc
      True  → loop (count + 1) (acc + (count * count))
```


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

- Repa, a parallel list processing library [11]

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

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

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]

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!

Questions?



- [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/>

- [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

- [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.

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