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

1

Outline

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.

$$map\ \_\ [\,] \qquad = [\,]$$
$$map\ f\ (x : xs) = f\ x : map\ f\ xs$$

$$sq\ x = x * x$$

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

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

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 *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 does the Prelude do better with the same functions?

4

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

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]

Transformations

```
┌─────────┐      ┌─────────┐      ┌─────────┐
│         │      │         │      │         │
│  A.hs   │ ───▶ │ A.core  │ ───▶ │   A.o   │
│         │      │         │      │         │
└─────────┘      └─────────┘      └─────────┘
  Haskell           Core           Bytecode
              Functional Language
```

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]

Transformations

| A.hs | A.core | A.o |

Haskell

Core
Functional Language

Bytecode
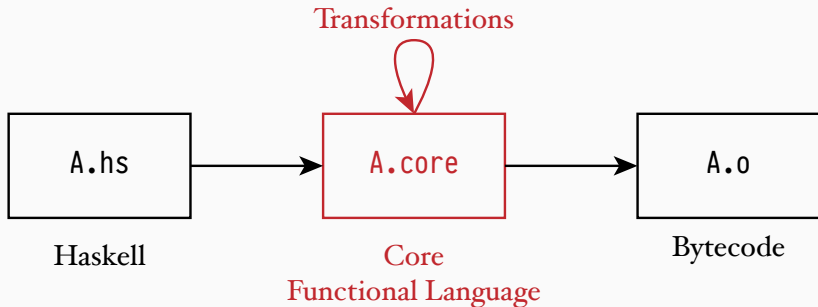
GHC performs several program transformations on Core to optimize the code

When GHC is given a Core program, it performs several types of transformations on the program. [6]

- Inlining functions
- 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. [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)
- 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. [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 #-}
  ```

  $x \nRightarrow id\,x$

- 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 #-}
```

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

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

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

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. [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.

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

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

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 #-}
```

$$foldr \ (+) \ 0 \ (build \ l) \equiv l \ (+) \ 0 \equiv 1 + (2 + (3 + 0))$$
$$\textbf{where}$$
$$l \ cons \ nil = 1 \ 'cons' \ (2 \ 'cons' \ (3 \ 'cons' \ nil))$$

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]

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

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

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

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

$$\equiv\ \{\text{-inline def of build -}\}$$

$$(\lambda c\ n \rightarrow foldr\ mapFB\ c\ f)\ n\ xs\ (:)\ [\,]$$

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

$$foldr\ (mapFB\ (:)\ f)\ [\,]\ xs$$

Manual rewrite rule application

Let's try applying the rewrite rules manually.

$sum\ (map\ sq\ xs)$

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

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

$\equiv$ { expand sum }

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

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

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

$\equiv$ { apply lambda }

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

Applying foldr: the empty case

We now look at empty case

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

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

$$0$$

Applying foldr: the list case

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

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

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

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

$\equiv$ {-apply lambda -}

$sq\ x + process\ xs$

$\equiv$ {-inline sq -}

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

Bringing both cases back together

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

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

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

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

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.

$$process_{fuse} \; [] \qquad = 0$$
$$process_{fuse} \; (x : xs) = x * x + process_{fuse} \; xs$$

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

Function	Time (ms)	Memory (MB)
$process$	41.86	265.26
$process'$	25.31	96.65
$process_{hand}$	25.31	96.65
$process_{fuse}$	25.31	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 $filter$.
- 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]

> **data** *Stream a* **where**
>   *Stream* :: $(s \rightarrow Step\ a\ s) \rightarrow s \rightarrow Stream\ a$

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.

$$stream :: [a] \to Stream\ a$$
$$stream\ xs = Stream\ uncons\ xs$$
> **where**
> $$uncons\ [\,] \qquad = Done$$
> $$uncons\ (x : xs) = Yield\ x\ xs$$

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

$unstream :: Stream\ a \rightarrow [\,a\,]$
$unstream\ (Stream\ next\ s0) = unfold\ next\ s0$
   where
     $unfold\ next\ s =$ **case** $next\ s$ **of**
        $Done \quad\ \rightarrow [\,]$
        $Skip\ s' \quad \rightarrow \quad\ unfold\ next\ s'$
        $Yield\ x\ s' \rightarrow x : unfold\ next\ s'$

Let's define *map* for *Streams*

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

$$map_s :: (a \rightarrow b) \rightarrow Stream\ a \rightarrow Stream\ b$$
$$map_s\ f\ (Stream\ next0\ s0) = Stream\ next\ s0$$
$\quad$ **where**
$\qquad next\ s = \textbf{case}\ next0\ s\ \textbf{of}$
$\qquad\quad Done \qquad \rightarrow Done$
$\qquad\quad Skip\ s' \quad \rightarrow Skip \qquad s'$
$\qquad\quad Yield\ x\ s' \rightarrow Yield\ (f\ x)\ s'$

$$map_{[a]} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$$
$$map_{[a]}\ f = unstream \circ map_s\ f \circ stream$$

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 #-}
```

$mapTestStream :: [Int] \rightarrow [Int]$

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

$map_{[a]}\ (+1) \circ map_{[a]}\ (*2)$

$\equiv$ {-expand mapl -}

$unstream \circ map_s\ (+1) \circ stream \circ unstream \circ map_s\ (*2) \circ stream$

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

$unstream \circ map_s\ (+1) \circ map_s\ (*2) \circ stream$

Map fused by Stream Fusion

Our map example

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

gets fused into this result.

$$mapTestStreamCompiled :: [Int] \rightarrow [Int]$$
$$mapTestStreamCompiled \ [\,] = [\,]$$
$$mapTestStreamCompiled \ (x : xs) =$$
$$\quad 1 + (x * 2) : mapTestStreamCompiled \ xs$$

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.

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

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)
$process$	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!*

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

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

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.

$$processVecGHC\ n = loop\ 1\ 0$$
 where
  $loop\ count\ acc = $ **case** $count \leqslant n$ **of**
   $False \rightarrow acc$
   $True \rightarrow loop\ (count + 1)\ (acc + (count * count))$

Other use cases for fusion

Besides vector, stream 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.

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.

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