Deforestation and Program Fusion: Applying Equational Transforms to Automatically Simplify Programs

github.com/ryanorendorff/xxxxxxxxxxxxx

Ryan Orendorff, PhD?[1]

May 2017

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

Table of Contents

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.

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

Where we have defined the functions as follows.

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

$$sq\ x = x * x$$

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

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

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.

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

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 sum and map.

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

Function	Time (ms)	Memory (MB)
$process$	220.0	265.26
$process'$	5.1	80.00

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.

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

Function	Time (ms)	Memory (MB)
$process$	220.0	265.26
$process'$	5.1	80.00

How does the Prelude do so much better with the same functions?

Let's try to optimize with an accumulator

What if we try to optimize $process$ using an accumulator?

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

Let's try to optimize with an accumulator

What if we try to optimize $process$ using an accumulator?

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

This means we make a tail-recursive "loop".

$$processFused :: [Int] \rightarrow Int$$
$$processFused = loop\ 0$$
$$\quad \mathbf{where}$$
$$\qquad loop\ n\ [\,] \qquad = n$$
$$\qquad loop\ n\ (x : xs) = loop\ (n + x * x)\ xs$$

Let's try to optimize with an accumulator

What if we try to optimize $process$ using an accumulator?

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

This means we make a tail-recursive "loop".

$$\begin{aligned}
&processFused :: [Int] \to Int \\
&processFused = loop\ 0 \\
&\quad \textbf{where} \\
&\quad\quad loop\ n\ [\,] \qquad\quad = n \\
&\quad\quad loop\ n\ (x : xs) = loop\ (n + x * x)\ xs
\end{aligned}$$

This mirrors the version of the program one would write imperatively.

How well does our manually fused version do?

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

$$process'\ xs = Prelude.sum \circ Prelude.map\ sq\ \$\ xs$$

$$processFused :: [Int] \to Int$$

$$processFused = loop\ 0$$

$$\quad \textbf{where}$$

$$\qquad loop\ n\ [\,] \qquad\quad = n$$

$$\qquad loop\ n\ (x : xs) = loop\ (n + x * x)\ xs$$

How well does our manually fused version do?

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

$$process'\ xs = Prelude.sum \circ Prelude.map\ sq\ \$\ xs$$

$$processFused :: [Int] \to Int$$
$$processFused = loop\ 0$$
$$\quad \textbf{where}$$
$$\qquad loop\ n\ [\,] \qquad = n$$
$$\qquad loop\ n\ (x : xs) = loop\ (n + x * x)\ xs$$

Function	Time (ms)	Memory (MB)
process	220.0	265.26
process′	5.1	80.00
processFused	4.7	80.00
process.c	2.6	8×10^{-5}

GHC generated the loop version automatically

Our manual version $processFused$.

$$
\begin{aligned}
&processFused :: [Int] \to Int\\
&processFused = loop\ 0\\
&\quad \mathbf{where}\\
&\qquad loop\ n\ [\,] \qquad\quad = n\\
&\qquad loop\ n\ (x : xs) = loop\ (n + x * x)\ xs
\end{aligned}
$$

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

$$
\begin{aligned}
&process' :: [Int] \to Int\\
&process'\ xs = loop'\ xs\ 0
\end{aligned}
$$

$$
\begin{aligned}
&loop' :: [Int] \to Int \to Int\\
&loop'\ [\,] \qquad\quad n = n\\
&loop'\ (x : xs)\ n = loop'\ xs\ (n + x * x)
\end{aligned}
$$

We want simpe/readible programs that are also performant

Our original version was simple to understand, but not fast.

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

Our accumulator one is fast but not simple.

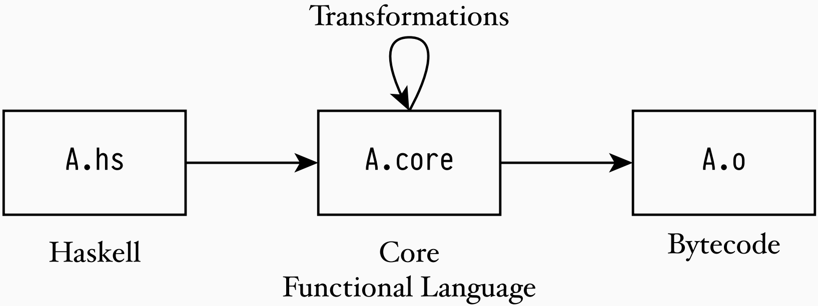
$$\begin{aligned}
&processFused :: [Int] \to Int \\
&processFused = loop\ 0 \\
&\quad \textbf{where} \\
&\qquad loop\ n\ [\,] \qquad\quad = n \\
&\qquad loop\ n\ (x : xs) = loop\ (n + x * x)\ xs
\end{aligned}$$

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

Table of Contents

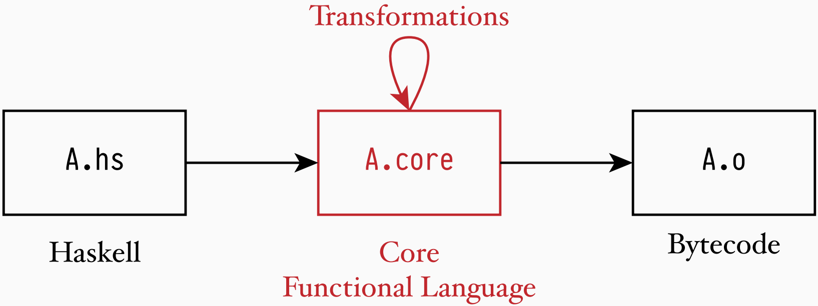
The GHC Compilation Pipeline converts Haskell into an intermediate language and then bytecode

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
- Removing redundant lambdas

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
- Removing redundant lambdas
- 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
- Removing redundant lambdas
- Simplifying constant expressions ($(x + 8) - 1$)
- Combining type casts

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
- Removing redundant lambdas
- Simplifying constant expressions ($(x + 8) - 1$)
- Combining type casts
- *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
- Removing redundant lambdas
- Simplifying constant expressions ($(x + 8) - 1$)
- Combining type casts
- *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 #-}

"Any time we see the term $id\ x$, replace it with 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 #-}
```

Rules have some restrictions

Rewrite rules have some gotchas.

- Rules doesn't prevent you from doing something silly

  $$\{\text{-\# RULES "id5" forall x. id x} = 5 \text{ \#-}\}$$

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

  $$\{\text{-\# RULES "id" forall x. id x} = \text{x \#-}\}$$

  $$x \not\Rightarrow id\ x$$

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 #-}

  $x \nRightarrow id\ x$

- 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.

- 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 will randomly choose one.

We can combine maps to traverse a list once

Let us introduce the following rule about maps.

$$\{\text{-\# RULES "map}_{\mathsf{f}}\text{/map}_{\mathsf{f}}\text{" forall f g xs. map}_{\mathsf{f}}\text{ f (map}_{\mathsf{f}}\text{ g xs)} = \text{map}_{\mathsf{f}}$$

We can combine maps to traverse a list once

Let us introduce the following rule about maps.

{-# RULES "$\mathsf{map_f}$/$\mathsf{map_f}$" forall f g xs. $\mathsf{map_f}$ f ($\mathsf{map_f}$ g xs) = $\mathsf{map_f}$

$$mapTest0 :: [Int] \to [Int]$$
$$mapTest0\ xs = map\ (+1)\ (map\ (*2)\ xs)$$

$$mapTest1 :: [Int] \to [Int]$$
$$mapTest1\ xs = map_f\ (+1)\ (map_f\ (*2)\ xs)$$

Our map fusion performs (a bit) better!

We can test our functions on a million elements

$$mapTest0\ xs = map\ (+1)\ (map\ (*2)\ xs)$$
$$mapTest0\ [1\,..\,1{,}000{,}000]$$

$$mapTest1\ xs = map_f\ (+1)\ (map_f\ (*2)\ xs)$$
$$mapTest1\ [1\,..\,1{,}000{,}000]$$

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

Function	Time (ms)	Memory (MB)
$mapTest0$	26.4	256.00
$mapTest1$	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

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

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

$$\begin{aligned}
&processFused :: [Int] \to Int \\
&processFused = loop\ 0 \\
&\quad \textbf{where} \\
&\qquad loop\ n\ [\,] \qquad\quad = n \\
&\qquad loop\ n\ (x : xs) = loop\ (n + x * x)\ xs
\end{aligned}$$

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

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

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

GHC accomplishes fusion with two functions: foldr and build.

$foldr$ combines the elements of a list

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

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

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

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

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 "$build/foldr$" forall f z (g :: forall b. (a -¿ b -¿ b) -¿

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

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.

$$map_{fb} :: (b \to l \to l) \to (a \to b) \to a \to l \to l$$
$$map_{fb}\ cons\ f = \lambda x\ ys \to f\ x\ \text{`}cons\text{`}\ ys$$

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.

$$map_{fb} :: (b \to l \to l) \to (a \to b) \to a \to l \to l$$
$$map_{fb}\ cons\ f = \lambda x\ ys \to f\ x\ \text{`}cons\text{`}\ ys$$

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

{-# RULES "map" forall f xs. map f xs $=$ $build$ (ņ -¿ $foldr$ (map

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.

$$map_{fb} :: (b \to l \to l) \to (a \to b) \to a \to l \to l$$
$$map_{fb}\ cons\ f = \lambda x\ ys \to f\ x\ `cons`\ ys$$

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

{-# RULES "map" forall f xs. map f xs $= build$ (ņ -¿ $foldr\ (map$

We also provide a way to combine sequential map_{fb} functions.

{-# RULES "mapFB" forall c f g. mapFB (mapFB c f) g = mapFI

This is where the example is expanded completely

Will show how the map sum example is converted into the loop we saw.

Will need

$$
\begin{aligned}
&foldr :: (a \to b \to b) \to b \to [a] \to b \\
&foldr\ f\ z = go \\
&\quad \textbf{where} \\
&\qquad go\ [\,] = z \\
&\qquad go\ (y : ys) = y\ `f`\ go\ ys
\end{aligned}
$$

{-# INLINE [0] foldr2 #-}

Table of Contents

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.

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

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

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

But has incredible performance!

Function	Time (ms)	Memory (MB)
$process$	220.0	265.26
$process'$	5.1	80.00
$processFused$	4.7	80.00
process.c	2.6	8×10^{-5}
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 ended up generating the following Core code.

$$
\begin{array}{l}
val\_\$s\$wfoldlM'\_loop \\
val\_\$s\$wfoldlM'\_loop = \\
\quad \lambda sc\ sc1 \rightarrow \\
\qquad \textbf{case}\ tagToEnum\#\ (<=\#sc\ 100000000\#)\ \textbf{of}\ \_\ \{ \\
\qquad\quad False \rightarrow sc1; \\
\qquad\quad True \rightarrow val\_\$s\$wfoldlM'\_loop\ (+\#sc\ 1\#)\ (+\#sc1\ (*\#sc \\
\qquad \}
\end{array}
$$

simplify this core, ignore unboxing

Repa: A numerical Haskell Library using Fusion

Repa also uses fusion in order to handle array operations.

Data Parallel Haskell: Nested Data Parallelism made easy

$$processDPH :: [:Int:] \to Int$$
$$processDPH = sumDPH \circ mapDPH\ sq \,\$\, xs$$

Does dispatch by MPI.