

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]

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

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] → Int

process xs = sum ∘ 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)
<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.

$process :: [Int] \rightarrow Int$

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

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

$process_{\text{hand}} [] = 0$

$process_{\text{hand}} (x : xs) = x * x + process_{\text{hand}} xs$

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

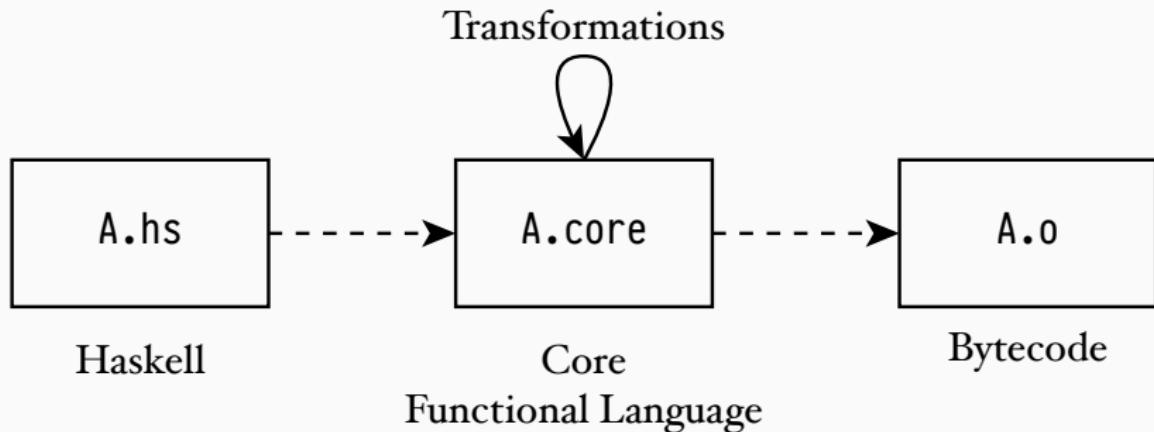
List fusion with *foldr/build*

Stream Fusion

Applications of Fusion

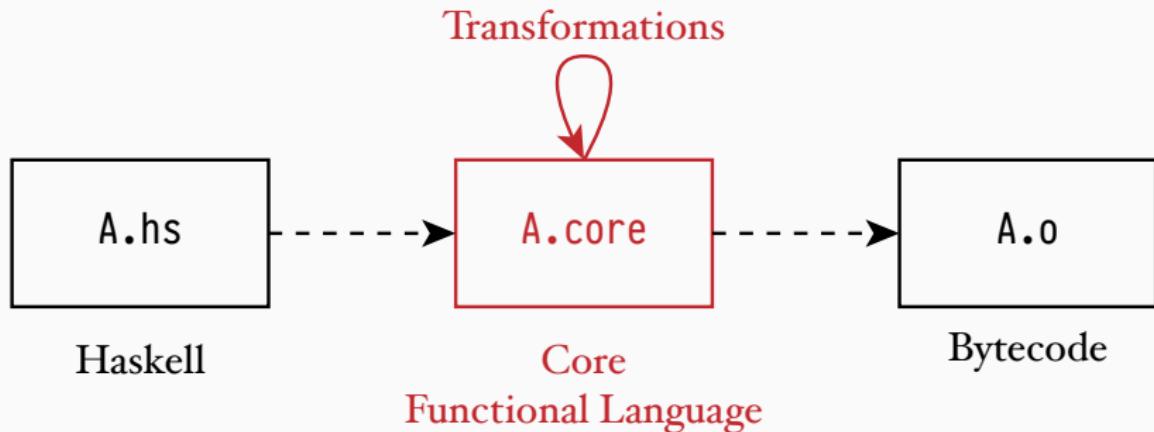
The GHC Compilation Pipeline converts Haskell into an intermediate language and then bytecode

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
- 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 #-}
```

- "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 #-}
```

- 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.  
  mapfuse f (mapfuse g xs) = mapfuse (f . g) xs #-}
```

$mapTest :: [Int] \rightarrow [Int]$

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

$mapTest_{\text{fuse}} :: [Int] \rightarrow [Int]$

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

Our map fusion performs (a bit) better!

We can test our functions on a million elements

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

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.

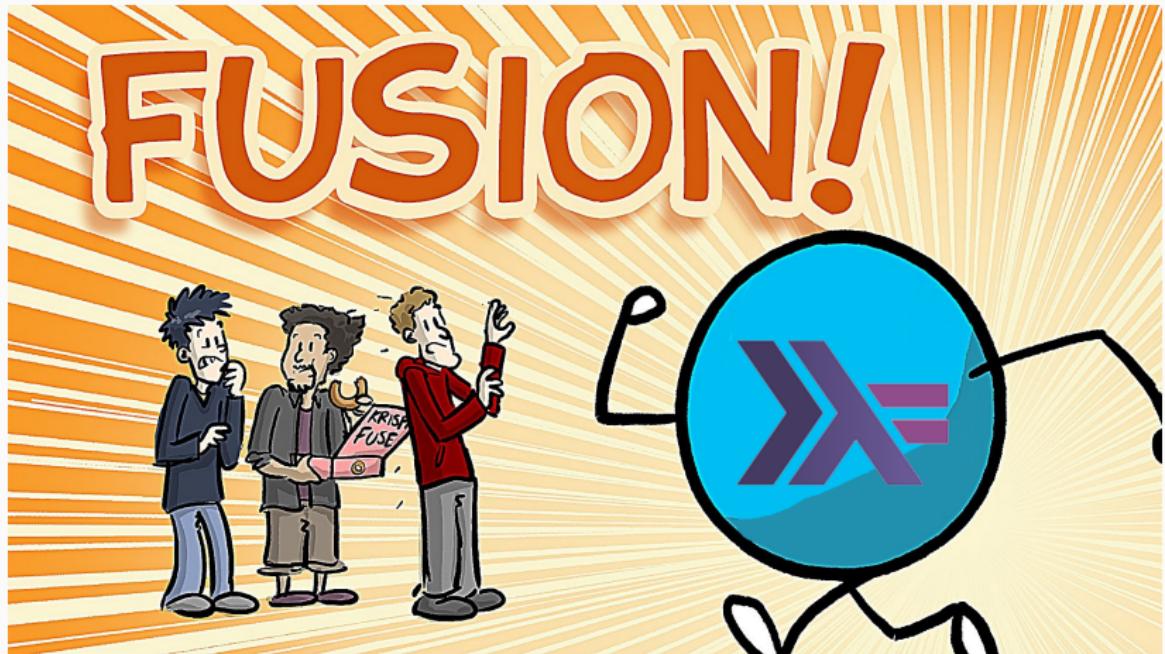
In our *process* function, we create an intermediate list

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

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

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

FUSION!



From PhD Comics

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 combines the elements of a list

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

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

$$\text{build} :: (\forall b. (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow b) \rightarrow [a]$$
$$\text{build } g = g \ (:) \ []$$
$$\text{build1 } l \equiv [1, 2, 3]$$

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

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

where

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

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]

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

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

$$\lambda x \ ys \rightarrow sq \ x : ys$$

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 ( $\lambda c n \rightarrow \text{foldr } \text{mapFB } c f) n xs)$   
≡ {-inline def of build -}  
 $(\lambda c n \rightarrow \text{foldr } \text{mapFB } c f) n xs) () []$   
≡ {-remove lambda -}  
foldr (mapFB (:)) f [] xs
```

Manual rewrite rule application

Let's try applying the rewrite rules manually.

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

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

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

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

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

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

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

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

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

Applying foldr: the empty case

We now look at empty case

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

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

0

Applying foldr: the list case

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

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

Bringing both cases back together

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

$$process_{fuse} [] = 0$$

$$process_{fuse} (x : xs) = x * x + process_{fuse} xs$$

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.

$$\text{process}_{\text{fuse}} [] = 0$$

$$\text{process}_{\text{fuse}} (x : xs) = x * x + \text{process}_{\text{fuse}} xs$$

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*.
- 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 → Step a s) → s → 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] → Stream a

stream xs = Stream uncons xs

where

uncons [] = Done

uncons (y : ys) = Yield y 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.

unstream :: Stream a → [a]

unstream (Stream next s0) = unfold next s0

where

unfold next s = case next s of

Done → []

Skip s' → unfold next s'

Yield x s' → x : unfold next s'

Let's define *map* for Streams

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

Let's try *map*.

$$map_s :: (a \rightarrow b) \rightarrow Stream\ a \rightarrow Stream\ b$$

$$map_s\ f\ (Stream\ next0\ s0) = Stream\ next\ s0$$

where

$$next\ s = \mathbf{case}\ next0\ s\ \mathbf{of}$$

$$Done \rightarrow Done$$

$$Skip\ s' \rightarrow Skip\ s'$$

$$Yield\ x\ s' \rightarrow Yield\ (f\ x)\ s'$$

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

$$map_{[a]}\ f = unstream \circ map_s\ f \circ stream$$

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 #-}
```

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

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

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

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

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

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

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

Map fused by Stream Fusion

Our map example

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

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

gets fused into this result.

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

$\text{mapTestStreamCompiled } [] = []$

$\text{mapTestStreamCompiled } (x : xs) =$

$1 + (x * 2) : \text{mapTestStreamCompiled } xs$

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.

```
process xs = sum ∘ 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>processfuse</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!

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.

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

And the final code generated is the following.

$$\text{processVecGHC } n = \text{loop } 1 \ 0$$

where

$$\text{loop } count \ acc = \text{case } count \leq n \ \text{of}$$
$$\text{False} \rightarrow acc$$
$$\text{True} \rightarrow \text{loop } (count + 1) (acc + (count * count))$$

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!

Questions?



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: https://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](http://www.haskellforall.com/2014/01/stream-fusion-for-pipes.html)