Shortcut Fusion in Haskell

Thomas Harper
tom.harper@cs.ox.ac.uk

Department of Computer Science
University of Oxford

Applied Functional Programming Summer School 2011

What is fusion?

What is fusion?

Example: *sumSq*

$$
\begin{aligned}
&\mathit{sumSq} :: \mathit{Int} \to \mathit{Int}\\
&\mathit{sumSq}\ y = \mathit{sum}\ (\mathit{map}\ \mathit{square}\ [1\,..\,y])\\
&\quad \mathbf{where}\\
&\qquad \mathit{square} :: \mathit{Int} \to \mathit{Int}\\
&\qquad \mathit{square}\ x = x * x
\end{aligned}
$$

Example: *sumSq*

$$
\begin{aligned}
&\mathit{sumSq} :: \mathit{Int} \rightarrow \mathit{Int}\\
&\mathit{sumSq}\ y = \mathit{sum}\ (\mathit{map}\ \mathit{square}\ [1\mathinner{..}y])\\
&\quad \textbf{where}\\
&\qquad \mathit{square} :: \mathit{Int} \rightarrow \mathit{Int}\\
&\qquad \mathit{square}\ x = x * x
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{sumSq}\ 5\\
&\mathit{sum}\ (\mathit{map}\ \mathit{square}\ [1, 2, 3, 4, 5])\\
&\mathit{sum}\ [1, 4, 9, 16, 25]\\
&55
\end{aligned}
$$

- Allocating these lists consumes memory, even though they do not appear in the result.

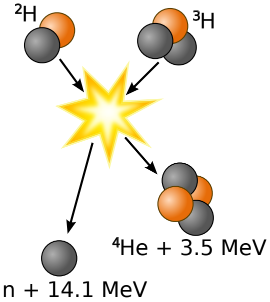
Intermediate data structures

- Allocating these lists consumes memory, even though they do not appear in the result.
- Such lists are called *intermediate data structures*.

Eliminating intermediate data structures

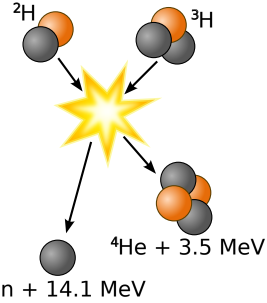
$$
\begin{array}{l}
\mathit{sumSq}' :: \mathit{Int} \to \mathit{Int} \\
\mathit{sumSq}'\ y = \mathit{go}\ 1 \\
\quad \textbf{where} \\
\qquad \mathit{go}\ i = \textbf{if}\ i > y \\
\qquad\qquad\quad \textbf{then}\ 0 \\
\qquad\qquad\quad \textbf{else}\ (\mathit{square}\ i) + \mathit{go}\ (i + 1)
\end{array}
$$

So, fusion is not so much this

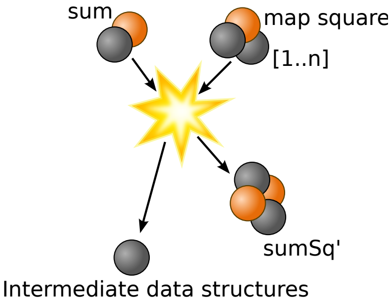


Eliminating intermediate data structures

So, fusion is not so much this



but more



Eliminating intermediate data structures

$$
\begin{array}{l}
sumSq' :: Int \rightarrow Int \\
sumSq'\ y = go\ 1 \\
\quad \textbf{where} \\
\qquad go\ i = \textbf{if}\ i > y \\
\qquad\qquad\quad \textbf{then}\ 0 \\
\qquad\qquad\quad \textbf{else}\ (square\ i) + go\ (i + 1)
\end{array}
$$

- No modularity
- Less clear
- Less maintainable

The Goal

We would like to write *sumSq*, and have the compiler would produce *sumSq′* automatically.

The Problem

Fusion involves inlining recursive functions.

The Problem

Fusion involves inlining recursive functions.

- This is really hard.

The Problem

Fusion involves inlining recursive functions.

- This is really hard.
- But GHC is already really good at inlining *non-recursive* functions...

Main Idea

Focus on fusing a small set of functions that encapsulate the recursion.

A first attempt

$$
\begin{aligned}
map\ f\ (map\ g\ xs) &= map\ (f \circ g)\ xs \\
filter\ p\ (filter\ q\ xs) &= filter\ (p \wedge q)\ xs
\end{aligned}
$$

A first attempt

$$\begin{aligned} map\ f\ (map\ g\ xs)\ &= map\ (f \circ g)\ xs \\ filter\ p\ (filter\ q\ xs) &= filter\ (p \wedge q)\ xs \end{aligned}$$

We can teach GHC to do this for us:

A first attempt

- What about *map f* (*filter p*) *xs*?

A first attempt

- What about $map\ f\ (filter\ p)\ xs$?

$$
\begin{aligned}
&mapFilter :: (a \to b) \to (a \to Bool) \to [a] \to [b]\\
&mapFilter\ f\ p\ [\,] \qquad\quad = [\,]\\
&mapFilter\ f\ p\ (x : xs) = \mathbf{if}\ p\ x\\
&\qquad\qquad\qquad\qquad\quad\ \mathbf{then}\ f\ x : mapFilter\ f\ p\ xs\\
&\qquad\qquad\qquad\qquad\quad\ \mathbf{else}\ mapFilter\ f\ p\ xs
\end{aligned}
$$

A first attempt

- What about $map\ f\ (filter\ p)\ xs$?

$$
\begin{aligned}
&mapFilter :: (a \to b) \to (a \to Bool) \to [a] \to [b] \\
&mapFilter\ f\ p\ [\,] \qquad\quad = [\,] \\
&mapFilter\ f\ p\ (x : xs) = \textbf{if}\ p\ x \\
&\qquad\qquad\qquad\qquad\quad \textbf{then}\ f\ x : mapFilter\ f\ p\ xs \\
&\qquad\qquad\qquad\qquad\quad \textbf{else}\ mapFilter\ f\ p\ xs
\end{aligned}
$$

$$map\ f\ (filter\ p\ xs) = mapFilter\ f\ p\ xs$$

A first attempt

- What about $map\ f\ (filter\ p)\ xs$?

$$
\begin{aligned}
&mapFilter :: (a \to b) \to (a \to Bool) \to [a] \to [b] \\
&mapFilter\ f\ p\ [\,] \qquad\quad = [\,] \\
&mapFilter\ f\ p\ (x : xs) = \mathbf{if}\ p\ x \\
&\qquad\qquad\qquad\qquad\quad\ \ \mathbf{then}\ f\ x : mapFilter\ f\ p\ xs \\
&\qquad\qquad\qquad\qquad\quad\ \ \mathbf{else}\ mapFilter\ f\ p\ xs
\end{aligned}
$$

$$map\ f\ (filter\ p\ xs) = mapFilter\ f\ p\ xs$$

- Okay, what about $filter\ p\ (map\ f\ xs)$?

A first attempt

- What about $map\ f\ (filter\ p)\ xs$?

$$
\begin{aligned}
&mapFilter :: (a \to b) \to (a \to Bool) \to [a] \to [b] \\
&mapFilter\ f\ p\ [\,] \qquad\quad = [\,] \\
&mapFilter\ f\ p\ (x : xs) = \mathbf{if}\ p\ x \\
&\qquad\qquad\qquad\qquad\quad\ \mathbf{then}\ f\ x : mapFilter\ f\ p\ xs \\
&\qquad\qquad\qquad\qquad\quad\ \mathbf{else}\ mapFilter\ f\ p\ xs
\end{aligned}
$$

$$map\ f\ (filter\ p\ xs) = mapFilter\ f\ p\ xs$$

- Okay, what about $filter\ p\ (map\ f\ xs)$?
- What happens if we add another function to the API? What happens when we try and fuse a longer pipeline? Combinatorial explosion!

Main Idea 2.0

Use one rule to fuse everything!

Encapsulating recursion

Many functions can be defined using *foldr*:

$$
\begin{aligned}
&\mathit{foldr} :: (a \to b \to b) \to b \to [a] \to b \\
&\mathit{foldr}\ c\ n\ [\,] \qquad\quad = n \\
&\mathit{foldr}\ c\ n\ (x : xs) = c\ x\ (\mathit{foldr}\ c\ n\ xs)
\end{aligned}
$$

Defining functions with *foldr*

$$
\begin{aligned}
\mathit{map}\ f\ \mathit{xs} &= \mathit{foldr}\ (\lambda a\ b \rightarrow f\ a : b)\ [\,]\ \mathit{xs} \\
\mathit{sum}\ \mathit{xs} &= \mathit{foldr}\ (+)\ 0\ \mathit{xs} \\
\mathit{filter}\ p\ \mathit{xs} &= \mathit{foldr}\ (\lambda a\ b \rightarrow \textbf{if}\ p\ a\ \textbf{then}\ a : b\ \textbf{else}\ b)\ [\,]\ \mathit{xs} \\
\mathit{xs} +\!\!+\ \mathit{ys} &= \mathit{foldr}\ (:)\ \mathit{ys}\ \mathit{xs} \\
\mathit{product}\ \mathit{xs} &= \mathit{foldr}\ (*)\ 1\ \mathit{xs}
\end{aligned}
$$

Fusing *foldr*

$$\begin{aligned} map\ f\ xs &= foldr\ (\lambda a\ b \to f\ a : b)\ [\,]\ xs \\ sum\ xs &= foldr\ (+)\ 0\ xs \end{aligned}$$

Fusing *foldr*

$$
\begin{aligned}
\mathit{map}\ f\ \mathit{xs} &= \mathit{foldr}\ (\lambda a\ b \rightarrow f\ a : b)\ [\,]\ \mathit{xs} \\
\mathit{sum}\ \mathit{xs} &= \mathit{foldr}\ (+)\ 0\ \mathit{xs}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{sum}\ (\mathit{map}\ \mathit{square}\ \mathit{xs}) \\
&\mathit{foldr}\ (+)\ 0\ (\mathit{foldr}\ (\lambda a\ b \rightarrow \mathit{square}\ a : b)\ [\,])
\end{aligned}
$$

Fusing *foldr*

$$
\begin{aligned}
map\ f\ xs &= foldr\ (\lambda a\ b \rightarrow f\ a : b)\ [\,]\ xs \\
sum\ xs &= foldr\ (+)\ 0\ xs
\end{aligned}
$$

$$
\begin{aligned}
&sum\ (map\ square\ xs) \\
&foldr\ (+)\ 0\ (foldr\ (\lambda a\ b \rightarrow square\ a : b)\ [\,])
\end{aligned}
$$

- Still not clear how to automatically rewrite this.
- Can see how the lists are consumed, but not how they are built.

Abstracting construction

We can abstract away (:) and []:

$$map\ f\ xs = (\lambda c\ n \rightarrow foldr\ (\lambda a\ b \rightarrow c\ (f\ a)\ b)\ n\ xs)\ (:)\ [\,]$$

Abstracting construction

We can abstract away (:) and []:

$$map\ f\ xs = (\lambda c\ n \rightarrow foldr\ (\lambda a\ b \rightarrow c\ (f\ a)\ b)\ n\ xs)\ (:)\ [\,]$$

And then abstract the list construction into a function:

$$build\ g = g\ (:)\ [\,]$$
$$map\ f\ xs = build\ (\lambda c\ n \rightarrow foldr\ (\lambda a\ b \rightarrow c\ (f\ a)\ b)\ n\ xs)$$

A list is consumed with *foldr*, and produced with *build*

Building lists with *build*

We can define $[m\,..\,n]$ directly

$$[x\,..\,y] = go\ x$$
$$\quad \textbf{where}$$
$$\qquad go\ x = \textbf{if}\ x > y\ \textbf{then}\ [\,]\ \textbf{else}\ x : go\ (x + 1)$$

Building lists with *build*

We can define $[m \mathinner{..} n]$ directly

$$
\begin{aligned}
&[x \mathinner{..} y] = go\ x \\
&\quad \textbf{where} \\
&\qquad go\ x = \textbf{if}\ x > y\ \textbf{then}\ [\,]\ \textbf{else}\ x : go\ (x + 1)
\end{aligned}
$$

but also using *build*:

$$
\begin{aligned}
&enumFromTo\ x\ y = build\ (\lambda c\ n \to eftInt\ x\ y\ c\ n) \\
&\quad \textbf{where} \\
&\qquad eftInt\ x\ y\ c\ n = go\ x \\
&\qquad\quad \textbf{where} \\
&\qquad\qquad go\ x = \textbf{if}\ x > y\ \textbf{then}\ n\ \textbf{else}\ c\ x\ (go\ (x + 1))
\end{aligned}
$$

Fusing *foldr* and *build*

Returning to our example:

$$sumSq\ y = sum\ (map\ square\ [1\mathinner{..}y])$$

Fusing *foldr* and *build*

Returning to our example:

$$sumSq\ y = sum\ (map\ square\ [1 \mathinner{..} y])$$

Inline $[1 \mathinner{..} y]$

$$\begin{aligned}
&sumSq\ y = sum\ (map\ square\ (build\ (\lambda c\ n \rightarrow eftInt\ 1\ y\ c\ n)))\\
&\quad \textbf{where}\\
&\qquad eftInt\ x\ y\ c\ n = go\ x\\
&\qquad\quad \textbf{where}\\
&\qquad\qquad go\ x = \textbf{if}\ x > y\ \textbf{then}\ n\ \textbf{else}\ c\ x\ (go\ (x + 1))
\end{aligned}$$

Fusing *foldr* and *build*

Inline $[1\,..\,y]$

$$
\begin{array}{l}
sumSq\ y = sum\ (map\ square\ (build\ (\lambda c\ n \rightarrow eftInt\ 1\ y\ c\ n))) \\
\quad \textbf{where} \\
\quad\quad eftInt\ x\ y\ c\ n = go\ x \\
\quad\quad\quad \textbf{where} \\
\quad\quad\quad\quad go\ x = \textbf{if}\ x > y\ \textbf{then}\ n\ \textbf{else}\ c\ x\ (go\ (x + 1))
\end{array}
$$

Fusing *foldr* and *build*

Inline $[1\,..\,y]$

$$\begin{aligned}
&\mathit{sumSq}\ y = \mathit{sum}\ (\mathit{map}\ \mathit{square}\ (\mathit{build}\ (\lambda c\ n \rightarrow \mathit{eftInt}\ 1\ y\ c\ n)))\\
&\quad \textbf{where}\\
&\qquad \mathit{eftInt}\ x\ y\ c\ n = \mathit{go}\ x\\
&\qquad\quad \textbf{where}\\
&\qquad\qquad \mathit{go}\ x = \textbf{if}\ x > y\ \textbf{then}\ n\ \textbf{else}\ c\ x\ (\mathit{go}\ (x+1))
\end{aligned}$$

Inlining *map*

$$\begin{aligned}
&\mathit{sumSq}\ y =\\
&\quad \mathit{sum}\ (\mathit{build}_1\ (\lambda c_1\ n_1 \rightarrow (\mathit{foldr}_1\ (\lambda a_1\ b_1 \rightarrow c_1\ (\mathit{square}\ a_1)\ b_1)\ n_1\\
&\qquad (\mathit{build}_2\ (\lambda c_2\ n_2 \rightarrow \mathit{eftInt}\ 1\ y\ c_2\ n_2)))))\\
&\quad \textbf{where}\\
&\qquad \mathit{eftInt}\ x\ y\ c\ n = \mathit{go}\ x\\
&\qquad\quad \textbf{where}\\
&\qquad\qquad \mathit{go}\ x = \textbf{if}\ x > y\ \textbf{then}\ n\ \textbf{else}\ c\ x\ (\mathit{go}\ (x+1))
\end{aligned}$$

Fusing *foldr* and *build*

We can see syntactically when an intermediate data structure is created:

$$foldr\ c\ n\ (build\ g)$$

Fusing *foldr* and *build*

We can see syntactically when an intermediate data structure is created:

$$foldr\ c\ n\ (build\ g)$$

$$foldr\ c\ n\ (g\ (:)\ [\,])$$

- g builds a list by placing (:) [ ] appropriate, but *foldr* will just replace them with c and n

Fusing *foldr* and *build*

We can see syntactically when an intermediate data structure is created:

$$foldr\ c\ n\ (build\ g)$$

$$foldr\ c\ n\ (g\ (:)\ [\,])$$

- g builds a list by placing (:) [ ] appropriate, but *foldr* will just replace them with c and n
- Instead, we can just apply g directly to c and n:

$$foldr\ c\ n\ (build\ g) = g\ c\ n$$

- As long as c and n have non-recursive definitions, GHC does the rest!

Fusing *foldr* and *build*

$$
\begin{array}{l}
sumSq\ y = \\
\quad sum\ (build_1\ (\lambda c_1\ n_1 \rightarrow (foldr_1\ (\lambda a_1\ b_1 \rightarrow c_1\ (square\ a_1)\ b_1)\ n_1 \\
\quad\quad (build_2\ (\lambda c_2\ n_2 \rightarrow eftInt\ 1\ y\ c_2\ n_2))))) \\
\quad \textbf{where} \\
\quad\quad eftInt\ x\ y\ c\ n = go\ x \\
\quad\quad\quad \textbf{where} \\
\quad\quad\quad\quad go\ x = \textbf{if}\ x > y\ \textbf{then}\ n\ \textbf{else}\ c\ x\ (go\ (x+1))
\end{array}
$$

Now we can apply the rewrite rule

Fusing *foldr* and *build*

$$\begin{aligned}
&sumSq\ y = \\
&\quad sum\ (build_1\ (\lambda c_1\ n_1 \to (foldr_1\ (\lambda a_1\ b_1 \to c_1\ (square\ a_1)\ b_1)\ n_1 \\
&\qquad (build_2\ (\lambda c_2\ n_2 \to eftInt\ 1\ y\ c_2\ n_2))))) \\
&\quad \textbf{where} \\
&\qquad eftInt\ x\ y\ c\ n = go\ x \\
&\qquad\quad \textbf{where} \\
&\qquad\qquad go\ x = \textbf{if}\ x > y\ \textbf{then}\ n\ \textbf{else}\ c\ x\ (go\ (x+1))
\end{aligned}$$

Now we can apply the rewrite rule

$$\begin{aligned}
&sumSq\ y = sum\ (build_1\ (\lambda c_1\ n_1 \to \\
&\quad (\lambda c_2\ n_2 \to eftInt\ 1\ y\ c_2\ n_2)\ (\lambda a_1\ b_1 \to c_1\ (square\ a_1)\ b_1)\ n_1)) \\
&\quad \textbf{where} \\
&\qquad eftInt\ x\ y\ c\ n = go\ x \\
&\qquad\quad \textbf{where} \\
&\qquad\qquad go\ x = \textbf{if}\ x > y\ \textbf{then}\ n\ \textbf{else}\ c\ x\ (go\ (x+1))
\end{aligned}$$

Fusing *foldr* and *build*

$$
\begin{array}{l}
sumSq\ y = sum\ (build_1\ (\lambda c_1\ n_1 \rightarrow \\
\quad (\lambda c_2\ n_2 \rightarrow eftInt\ 1\ y\ c_2\ n_2)\ (\lambda a_1\ b_1 \rightarrow c_1\ (square\ a_1)\ b_1)\ n_1)) \\
\quad \textbf{where} \\
\qquad eftInt\ x\ y\ c\ n = go\ x \\
\qquad\quad \textbf{where} \\
\qquad\qquad go\ x = \textbf{if}\ x > y\ \textbf{then}\ n\ \textbf{else}\ c\ x\ (go\ (x+1))
\end{array}
$$

Fusing *foldr* and *build*

$$
\begin{aligned}
&\mathit{sumSq}\ y = \mathit{sum}\ (\mathit{build}_1\ (\lambda c_1\ n_1 \rightarrow \\
&\quad (\lambda c_2\ n_2 \rightarrow \mathit{eftInt}\ 1\ y\ c_2\ n_2)\ (\lambda a_1\ b_1 \rightarrow c_1\ (\mathit{square}\ a_1)\ b_1)\ n_1)) \\
&\quad \textbf{where} \\
&\qquad \mathit{eftInt}\ x\ y\ c\ n = \mathit{go}\ x \\
&\qquad\quad \textbf{where} \\
&\qquad\qquad \mathit{go}\ x = \textbf{if}\ x > y\ \textbf{then}\ n\ \textbf{else}\ c\ x\ (\mathit{go}\ (x+1))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{sumSq}\ y = \mathit{sum}\ (\mathit{build}_1\ (\lambda c_1\ n_1 \rightarrow \\
&\quad \mathit{eftInt}\ 1\ y\ (\lambda a_1\ b_1 \rightarrow c_1\ (\mathit{square}\ a_1)\ b_1)\ n_1) \\
&\quad \textbf{where} \\
&\qquad \mathit{eftInt}\ x\ y\ c\ n = \mathit{go}\ x \\
&\qquad\quad \textbf{where} \\
&\qquad\qquad \mathit{go}\ x = \textbf{if}\ x > y\ \textbf{then}\ n\ \textbf{else}\ c\ x\ (\mathit{go}\ (x+1))
\end{aligned}
$$

Fusing *foldr* and *build*

$$
\begin{array}{l}
sumSq\ y = sum\ (build_1\ (\lambda c_1\ n_1 \rightarrow \\
\quad eftInt\ 1\ y\ (\lambda a_1\ b_1 \rightarrow c_1\ (square\ a_1)\ b_1)\ n_1) \\
\quad \textbf{where} \\
\qquad eftInt\ x\ y\ c\ n = go\ x \\
\qquad\quad \textbf{where} \\
\qquad\qquad go\ x = \textbf{if}\ x > y\ \textbf{then}\ n\ \textbf{else}\ c\ x\ (go\ (x+1))
\end{array}
$$

Fusing *foldr* and *build*

$$
\begin{array}{l}
sumSq\ y = sum\ (build_1\ (\lambda c_1\ n_1 \rightarrow \\
\quad eftInt\ 1\ y\ (\lambda a_1\ b_1 \rightarrow c_1\ (square\ a_1)\ b_1)\ n_1) \\
\quad \textbf{where} \\
\quad\quad eftInt\ x\ y\ c\ n = go\ x \\
\quad\quad\quad \textbf{where} \\
\quad\quad\quad\quad go\ x = \textbf{if}\ x > y\ \textbf{then}\ n\ \textbf{else}\ c\ x\ (go\ (x+1))
\end{array}
$$

$$
\begin{array}{l}
sumSq\ y = foldr\ (+)\ 0\ (build_1\ (\lambda c_1\ n_1 \rightarrow \\
\quad eftInt\ 1\ y\ (\lambda a_1\ b_1 \rightarrow c_1\ (square\ a_1)\ b_1)\ n_1) \\
\quad \textbf{where} \\
\quad\quad eftInt\ x\ y\ c\ n = go\ x \\
\quad\quad\quad \textbf{where} \\
\quad\quad\quad\quad go\ x = \textbf{if}\ x > y\ \textbf{then}\ n\ \textbf{else}\ c\ x\ (go\ (x+1))
\end{array}
$$

Fusing *foldr* and *build*

$$
\begin{array}{l}
\mathit{sumSq}\ y = \mathit{foldr}\ (+)\ 0\ (\mathit{build}_1\ (\lambda c_1\ n_1 \rightarrow \\
\quad \mathit{eftInt}\ 1\ y\ (\lambda a_1\ b_1 \rightarrow c_1\ (\mathit{square}\ a_1)\ b_1)\ n_1) \\
\quad \textbf{where} \\
\qquad \mathit{eftInt}\ x\ y\ c\ n = \mathit{go}\ x \\
\qquad\quad \textbf{where} \\
\qquad\qquad \mathit{go}\ x = \textbf{if}\ x > y\ \textbf{then}\ n\ \textbf{else}\ c\ x\ (\mathit{go}\ (x+1))
\end{array}
$$

Fusing *foldr* and *build*

$$
\begin{aligned}
&sumSq\ y = foldr\ (+)\ 0\ (build_1\ (\lambda c_1\ n_1 \rightarrow\\
&\quad eftInt\ 1\ y\ (\lambda a_1\ b_1 \rightarrow c_1\ (square\ a_1)\ b_1)\ n_1)\\
&\quad \textbf{where}\\
&\qquad eftInt\ x\ y\ c\ n = go\ x\\
&\qquad\quad \textbf{where}\\
&\qquad\qquad go\ x = \textbf{if}\ x > y\ \textbf{then}\ n\ \textbf{else}\ c\ x\ (go\ (x+1))
\end{aligned}
$$

$$
\begin{aligned}
&sumSq\ y = (\lambda c_1\ n_1 \rightarrow\\
&\quad eftInt\ 1\ y\ (\lambda a_1\ b_1 \rightarrow c_1\ (square\ a_1)\ b_1)\ n_1)\ (+)\ 0\\
&\quad \textbf{where}\\
&\qquad eftInt\ x\ y\ c\ n = go\ x\\
&\qquad\quad \textbf{where}\\
&\qquad\qquad go\ x = \textbf{if}\ x > y\ \textbf{then}\ n\ \textbf{else}\ c\ x\ (go\ (x+1))
\end{aligned}
$$

Fusing *foldr* and *build*

$$
\begin{aligned}
&sumSq\ y = (\lambda c_1\ n_1 \rightarrow \\
&\quad eftInt\ 1\ y\ (\lambda a_1\ b_1 \rightarrow c_1\ (square\ a_1)\ b_1)\ n_1)\ (+)\ 0 \\
&\quad \textbf{where} \\
&\qquad eftInt\ x\ y\ c\ n = go\ x \\
&\qquad\quad \textbf{where} \\
&\qquad\qquad go\ x = \textbf{if}\ x > y\ \textbf{then}\ n\ \textbf{else}\ c\ x\ (go\ (x+1))
\end{aligned}
$$

Fusing *foldr* and *build*

$$
\begin{aligned}
&\mathit{sumSq}\ y = (\lambda c_1\ n_1 \rightarrow \\
&\quad \mathit{eftInt}\ 1\ y\ (\lambda a_1\ b_1 \rightarrow c_1\ (\mathit{square}\ a_1)\ b_1)\ n_1)\ (+)\ 0 \\
&\quad \textbf{where} \\
&\qquad \mathit{eftInt}\ x\ y\ c\ n = \mathit{go}\ x \\
&\qquad\quad \textbf{where} \\
&\qquad\qquad \mathit{go}\ x = \textbf{if}\ x > y\ \textbf{then}\ n\ \textbf{else}\ c\ x\ (\mathit{go}\ (x+1))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{sumSq}\ y = \mathit{eftInt}\ 1\ y\ (\lambda a_1\ b_1 \rightarrow (\mathit{square}\ a_1) + b_1)\ 0) \\
&\quad \textbf{where} \\
&\qquad \mathit{eftInt}\ x\ y\ c\ n = \mathit{go}\ x \\
&\qquad\quad \textbf{where} \\
&\qquad\qquad \mathit{go}\ x = \textbf{if}\ x > y\ \textbf{then}\ n\ \textbf{else}\ c\ x\ (\mathit{go}\ (x+1))
\end{aligned}
$$

Fusing *foldr* and *build*

$$
\begin{array}{l}
\mathit{sumSq}\ y = \mathit{eftInt}\ 1\ y\ (\lambda a_1\ b_1 \rightarrow (\mathit{square}\ a_1) + b_1)\ 0) \\
\quad \textbf{where} \\
\qquad \mathit{eftInt}\ x\ y\ c\ n = \mathit{go}\ x \\
\qquad\quad \textbf{where} \\
\qquad\qquad \mathit{go}\ x = \textbf{if}\ x > y\ \textbf{then}\ n\ \textbf{else}\ c\ x\ (\mathit{go}\ (x+1))
\end{array}
$$

Fusing *foldr* and *build*

$$
\begin{array}{l}
sumSq\ y = eftInt\ 1\ y\ (\lambda a_1\ b_1 \to (square\ a_1) + b_1)\ 0) \\
\quad \textbf{where} \\
\quad\quad eftInt\ x\ y\ c\ n = go\ x \\
\quad\quad\quad \textbf{where} \\
\quad\quad\quad\quad go\ x = \textbf{if}\ x > y\ \textbf{then}\ n\ \textbf{else}\ c\ x\ (go\ (x+1))
\end{array}
$$

$$
\begin{array}{l}
sumSq\ y = go\ 1 \\
\quad \textbf{where} \\
\quad\quad go\ x = \textbf{if}\ x > y \\
\quad\quad\quad\quad\quad\quad \textbf{then}\ 0 \\
\quad\quad\quad\quad\quad\quad \textbf{else}\ (\lambda a_1\ b_1 \to (square\ a_1) + b_1)\ x\ (go\ (x+1))
\end{array}
$$

Fusing *foldr* and *build*

$$
\begin{array}{l}
\mathit{sumSq}\ y = \mathit{go}\ 1 \\
\quad \textbf{where} \\
\qquad \mathit{go}\ x = \textbf{if}\ x > y \\
\qquad\qquad\qquad \textbf{then}\ 0 \\
\qquad\qquad\qquad \textbf{else}\ (\lambda a_1\ b_1 \rightarrow (\mathit{square}\ a_1) + b_1)\ x\ (\mathit{go}\ (x+1))
\end{array}
$$

Fusing *foldr* and *build*

$$
\begin{aligned}
&\mathit{sumSq}\ y = \mathit{go}\ 1\\
&\quad \textbf{where}\\
&\qquad \mathit{go}\ x = \textbf{if}\ x > y\\
&\qquad\qquad\quad \textbf{then}\ 0\\
&\qquad\qquad\quad \textbf{else}\ (\lambda a_1\ b_1 \rightarrow (\mathit{square}\ a_1) + b_1)\ x\ (\mathit{go}\ (x+1))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{sumSq}\ y = \mathit{go}\ 1\\
&\quad \textbf{where}\\
&\qquad \mathit{go}\ x = \textbf{if}\ x > y\\
&\qquad\qquad\quad \textbf{then}\ 0\\
&\qquad\qquad\quad \textbf{else}\ \mathit{square}\ x + \mathit{go}\ (x+1)
\end{aligned}
$$

Success!

- We now have a way to fuse a specific set of functions that transform some datatype

Success!

- We now have a way to fuse a specific set of functions that transform some datatype
- ... as long as they are folds.

The Good News

Lots of functions can be written using *foldr* and *build*.

The Good News

Lots of functions can be written using *foldr* and *build*.

The Bad News

Some really important ones do not play nice.

The issue of *foldl* and *zip* as folds

- Two important functions, *foldl* and *zip*, are not folds.
- We can smash them into a form that uses *foldr*, but the resulting performance is poor.

Dualising *foldr*/*build* fusion

There is a dual to *foldr*, called *unfoldr*

$$
\begin{aligned}
&unfoldr :: (s \to Maybe\ (a, s)) \to s \to [a] \\
&unfoldr\ step\ s = \textbf{case}\ step\ s\ \textbf{of} \\
&\quad Just\ (a, s') \to a : unfoldr\ step\ s' \\
&\quad Nothing \qquad \to [\,]
\end{aligned}
$$

Defining functions with *unfoldr*

$$
\begin{aligned}
&\mathit{mapS}\ f\ \mathit{xs} = \mathit{unfoldr}\ \mathit{step}\ \mathit{xs}\\
&\quad \mathbf{where}\\
&\qquad \mathit{step}\ [\,] \qquad\ \ = \mathit{Nothing}\\
&\qquad \mathit{step}\ (x : \mathit{xs}) = \mathit{Just}\ (f\ x, \mathit{xs})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{enumFromToS}\ m\ n = \mathit{unfoldr}\ \mathit{step}\ m\\
&\quad \mathbf{where}\\
&\qquad \mathit{step}\ x = \mathbf{if}\ x > n\\
&\qquad\qquad\quad\ \ \mathbf{then}\ \mathit{Nothing}\\
&\qquad\qquad\quad\ \ \mathbf{else}\ \mathit{Just}\ (x, x + 1)
\end{aligned}
$$

Fusing unfolds

- We can fuse unfolds, too.

Fusing unfolds

- We can fuse unfolds, too.
- First, we are going to tweak the presentation bit.

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

Fusing unfolds

- We can fuse unfolds, too.
- First, we are going to tweak the presentation bit.

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

$$\textbf{data}\ \mathit{CoList}\ a = \exists\, s.\ \mathit{CoList}\ (s \to \mathit{Step}\ a\ s)\ s$$

Unfolds with *CoList*

- A *CoList* is just a set of arguments for an unfold.

Unfolds with *CoList*

- A *CoList* is just a set of arguments for an unfold.

$$
\begin{array}{l}
\mathit{unfold} :: \mathit{CoList}\ a \to [a] \\
\mathit{unfold}\ (\mathit{CoList}\ \mathit{step}\ s) = \mathit{go}\ s \\
\quad \textbf{where} \\
\qquad \mathit{go}\ s = \textbf{case}\ \mathit{step}\ s\ \textbf{of} \\
\qquad\quad \mathit{Done} \qquad \to [\,] \\
\qquad\quad \mathit{Yield}\ a\ s' \to a : \mathit{go}\ s'
\end{array}
$$

Transforming *CoLists*

- We can define transformations from one *CoList* to another.

Transforming *CoLists*

- We can define transformations from one *CoList* to another.

$$
\begin{aligned}
&mapCL :: (a \to b) \to CoList\ a \to CoList\ b \\
&mapCL\ f\ (CoList\ step\ s) = CoList\ step'\ s \\
&\quad \mathbf{where} \\
&\qquad step'\ s = \mathbf{case}\ step\ s\ \mathbf{of} \\
&\qquad\quad Done \qquad\ \to Done \\
&\qquad\quad Yield\ a\ s' \to Yield\ (f\ a)\ s'
\end{aligned}
$$

Transforming *CoLists*

- We can define transformations from one *CoList* to another.

$$
\begin{array}{l}
mapCL :: (a \to b) \to CoList\ a \to CoList\ b \\
mapCL\ f\ (CoList\ step\ s) = CoList\ step'\ s \\
\quad \textbf{where} \\
\qquad step'\ s = \textbf{case}\ step\ s\ \textbf{of} \\
\qquad\quad Done \quad\ \ \to Done \\
\qquad\quad Yield\ a\ s' \to Yield\ (f\ a)\ s'
\end{array}
$$

$$
\begin{array}{l}
enumFromToCL :: Int \to Int \to CoList\ Int \\
enumFromToCL\ x\ y = CoList\ step\ x \\
\quad \textbf{where} \\
\qquad step\ x = \textbf{if}\ x > y \\
\qquad\qquad\qquad \textbf{then}\ Done \\
\qquad\qquad\qquad \textbf{else}\ Yield\ x\ (x + 1)
\end{array}
$$

Converting between Lists and *CoLists*

- We can write all our transformations over *CoList* and if they are non-recursive, they will fuse.
- If we want to get back to lists we just use *unfold*.
- We can also a list into a CoList:

$$
\begin{aligned}
&\mathit{destroy} :: [a] \rightarrow \mathit{CoList}\ a \\
&\mathit{destroy}\ \mathit{xs} = \mathit{CoList}\ \mathit{step}\ \mathit{xs} \\
&\quad \mathbf{where} \\
&\qquad \mathit{step}\ [\,] \qquad\quad = \mathit{Done} \\
&\qquad \mathit{step}\ (x : \mathit{xs}) = \mathit{Yield}\ x\ \mathit{xs}
\end{aligned}
$$

Converting between Lists and *CoLists*

- Using *unfold* and *destroy*, we can turn a *CoList* function into a list function.

Converting between Lists and *CoLists*

- Using *unfold* and *destroy*, we can turn a *CoList* function into a list function.

$$map\ f = unfold \circ mapCL\ f \circ destroy$$

Converting between Lists and *CoLists*

- Using *unfold* and *destroy*, we can turn a *CoList* function into a list function.

$$map\ f = unfold \circ mapCL\ f \circ destroy$$

- Suppose we have a similar definition for *filterCL* and *filter*.
- If we inline $(map\ f\ (filter\ p\ xs))$, we get

$$unfold \circ mapCL\ f \circ destroy \circ unfold \circ filterCL\ p \circ destroy$$

- As with *foldr* and *build*, we can "see" where the intermediate data structures are, and use a similar rewrite rule:

$$destroy\ (unfold\ xs) = xs$$

- As with *foldr* and *build*, we can "see" where the intermediate data structures are, and use a similar rewrite rule:

$$\mathit{destroy}\ (\mathit{unfold}\ \mathit{xs}) = \mathit{xs}$$

- And if we apply it, we can remove an intermediate data structure:

$$\mathit{unfold} \circ \mathit{mapCL}\ f \circ \mathit{filterCL}\ p \circ \mathit{destroy}$$

- We can define *filter* for *CoLists*:

$$
\begin{aligned}
&\mathit{filterCL} :: (a \to \mathit{Bool}) \to \mathit{CoList}\ a \to \mathit{CoList}\ a \\
&\mathit{filterCL}\ p\ (\mathit{CoList}\ \mathit{step}\ s) = \mathit{CoList}\ \mathit{step}'\ s \\
&\quad \mathbf{where} \\
&\qquad \mathit{step}'\ s = \mathbf{case}\ \mathit{step}\ s\ \mathbf{of} \\
&\qquad\quad \mathit{Done} \qquad\ \to \mathit{Done} \\
&\qquad\quad \mathit{Yield}\ a\ s' \to \mathbf{if}\ p\ a \\
&\qquad\qquad\qquad\qquad\ \ \mathbf{then}\ \mathit{Yield}\ a\ s' \\
&\qquad\qquad\qquad\qquad\ \ \mathbf{else}\ \mathit{step}'\ s'
\end{aligned}
$$

- Unfortunately, it breaks everything.

Stream Fusion

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

$$\textbf{data}\ \mathit{Stream}\ a = \exists\ s.\ \mathit{Stream}\ (s \to \mathit{Step}\ a\ s)\ s$$

filter with *Stream*

$$
\begin{array}{l}
\mathit{filterS} :: (a \to \mathit{Bool}) \to \mathit{Stream}\ a \to \mathit{Stream}\ a \\
\mathit{filterS}\ p\ (\mathit{Stream}\ \mathit{step}\ s) = \mathit{Stream}\ \mathit{step}'\ s \\
\quad \textbf{where} \\
\qquad \mathit{step}'\ s = \textbf{case}\ \mathit{step}\ s\ \textbf{of} \\
\qquad\quad \begin{array}{lcl}
\mathit{Done} & \to & \mathit{Done} \\
\mathit{Skip}\ s' & \to & \mathit{Skip}\ s' \\
\mathit{Yield}\ a\ s' & \to & \textbf{if}\ p\ a \\
 & & \textbf{then}\ \mathit{Yield}\ a\ s' \\
 & & \textbf{else}\ \mathit{Skip}\ s'
\end{array}
\end{array}
$$

Converting to and from *Stream*

- *stream* is the same as *destroy*

$$
\begin{aligned}
&\mathit{stream} :: [a] \rightarrow \mathit{Stream}\ a \\
&\mathit{stream}\ \mathit{xs} = \mathit{Stream}\ \mathit{step}\ \mathit{xs} \\
&\quad \textbf{where} \\
&\qquad \mathit{step}\ [\,] \qquad\quad = \mathit{Done} \\
&\qquad \mathit{step}\ (x : \mathit{xs}) = \mathit{Yield}\ x\ \mathit{xs}
\end{aligned}
$$

Converting to and from *Stream*

- *stream* is the same as *destroy*

$$
\begin{aligned}
&\mathit{stream} :: [a] \rightarrow \mathit{Stream}\ a\\
&\mathit{stream}\ \mathit{xs} = \mathit{Stream}\ \mathit{step}\ \mathit{xs}\\
&\quad \textbf{where}\\
&\qquad \mathit{step}\ [\,] = \mathit{Done}\\
&\qquad \mathit{step}\ (x : \mathit{xs}) = \mathit{Yield}\ x\ \mathit{xs}
\end{aligned}
$$

- *unstream* is almost the same

$$
\begin{aligned}
&\mathit{unstream} :: \mathit{Stream}\ a \rightarrow [a]\\
&\mathit{unstream}\ (\mathit{Stream}\ \mathit{step}\ s) = \mathit{go}\ s\\
&\quad \textbf{where}\\
&\qquad \mathit{go}\ s = \textbf{case}\ \mathit{step}\ s\ \textbf{of}\\
&\qquad\quad \mathit{Done} \qquad\ \rightarrow [\,]\\
&\qquad\quad \mathit{Skip}\ s' \qquad \rightarrow \mathit{go}\ s'\\
&\qquad\quad \mathit{Yield}\ a\ s' \rightarrow a : \mathit{go}\ s'
\end{aligned}
$$

zip and *foldl*

- Not only we can fuse *filter*, we can also efficiently *foldl* and *zip*.

zip and *foldl*

- Not only we can fuse *filter*, we can also efficiently *foldl* and *zip*.
- Reminder:

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

- Not only we can fuse *filter*, we can also efficiently *foldl* and *zip*.
- Reminder:

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

- *foldl* is an extremely efficient way to reduce a list, and we get the behaviour with streams:

$$\begin{aligned}
&\mathit{foldlS} :: (b \to a \to b) \to b \to \mathit{Stream}\ a \to b \\
&\mathit{foldlS}\ f\ z\ (\mathit{Stream}\ \mathit{step}\ s) = \mathit{go}\ z\ s \\
&\quad \textbf{where} \\
&\qquad \mathit{go}\ z\ s = \textbf{case}\ \mathit{step}\ s\ \textbf{of} \\
&\qquad\quad \mathit{Done} \qquad\ \to z \\
&\qquad\quad \mathit{Skip}\ s' \qquad \to \mathit{go}\ z\ s' \\
&\qquad\quad \mathit{Yield}\ a\ s' \to \mathit{go}\ (f\ z\ a)\ s'
\end{aligned}$$

zip and *foldl*

- *zip* takes advantage of another feature of unfolds, state

$$
\begin{array}{l}
zipS :: Stream\ a \rightarrow Stream\ b \rightarrow Stream\ (a, b) \\
zipS\ (Stream\ step1\ s1)\ (Stream\ step2\ s2) = \\
\quad\quad Stream\ step'\ (s1, s2, Nothing) \\
\quad \mathbf{where} \\
\quad\quad step'\ (s1, s2, Nothing) = \mathbf{case}\ step1\ s1\ \mathbf{of} \\
\quad\quad\quad \begin{array}{ll}
Done & \rightarrow Done \\
Skip\ s1' & \rightarrow Skip\ (s1', s2, Nothing) \\
Yield\ a\ s1' & \rightarrow Skip\ (s1', s2, Just\ a)
\end{array} \\
\quad\quad step'\ (s1', s2, Just\ a) = \mathbf{case}\ step2\ s2\ \mathbf{of} \\
\quad\quad\quad \begin{array}{ll}
Done & \rightarrow Done \\
Skip\ s2' & \rightarrow Skip\ (s1', s2', Just\ a) \\
Yield\ b\ s2' & \rightarrow Yield\ (a, b)\ (s1', s2', Nothing)
\end{array}
\end{array}
$$

Applications

- Stream fusion combines unfold fusion with a very elegrant presentation.
- It allows us to write fusible functions for any data structure that we can define a *stream* and *unstream* for.
- This is a huge win for arrays.
- Examples are *Data.Bytestring* and *Data.Text*

Generalising

- The notion of folds and unfolds are not unique to lists.
- Although a less researched area, it is possible to fuse functions over other datatypes.

Conclusions

- Shortcut fusion is a useful tool when trying to get good performance from a library.
- You take care of standardising the recursion, keeping the transformers non-recursive, and GHC will do the rest automagically.
- There is no ideal recursion scheme, what you choose depends on your API and data structure.