Lecture 4: Higher Order Functions

Søren Haagerup

Department of Mathematics and Computer Science
University of Southern Denmark, Odense

September 26, 2017

HIGHER ORDER FUNCTIONS

The order of a function

Definition

- A **first order** function is a function whose input and output are not functions.
- A function has order $n+1$ if its input or output contain a function of maximum order n
- A function with order $n \geqslant 2$ is a **higher order function**

The order of a function

- $f :: Num\ a \Rightarrow a \rightarrow a$
  $f\ x = x + 5$
  What order does this function have?
  - is a **first order** function, since neither the input nor the output of the function is a function.
- $add :: Num\ a \Rightarrow a \rightarrow (a \rightarrow a)$
  $add\ x\ y = x + y$
  What order does this function have?
  - It is a **second order** function, since it returns a first order function.
- $add' :: Num\ a \Rightarrow (a, a) \rightarrow a$
  $add'\ (x, y) = x + y$
  What order does this function have?
  - It is a **first order** function

Why are higher order functions useful?

Let's say that we want to sum all the even numbers in a list. If we have just learned about recursive list algorithms, we would likely end up with a function like

$$
\begin{aligned}
&\mathit{sumEvens} :: [\mathit{Int}] \rightarrow \mathit{Int} \\
&\mathit{sumEvens}\ [\,] && = 0 \\
&\mathit{sumEvens}\ (x : xs) \mid \mathit{even}\ x && = x + \mathit{sumEvens}\ xs \\
&\mathit{sumEvens}\ (x : xs) \mid \mathit{otherwise} && = \mathit{sumEvens}\ xs
\end{aligned}
$$

If later, we are told to sum all the odd numbers as well

$$
\begin{aligned}
&\mathit{sumOdds} :: [\mathit{Int}] \rightarrow \mathit{Int} \\
&\mathit{sumOdds}\ [\,] && = 0 \\
&\mathit{sumOdds}\ (x : xs) \mid \mathit{odd}\ x && = x + \mathit{sumOdds}\ xs \\
&\mathit{sumOdds}\ (x : xs) \mid \mathit{otherwise} && = \mathit{sumOdds}\ xs
\end{aligned}
$$

That's quite a lot of code for a simple problem!

Why are higher order functions useful?

A different approach is to use **higher order functions** from Haskell's standard library, to solve the problem. In this problem we use the higher order function *filter*:

$$
\begin{aligned}
&\mathit{sumEvens}', \mathit{sumOdds}' :: [\mathit{Int}] \rightarrow \mathit{Int} \\
&\mathit{sumEvens}' = \mathit{sum}\ (\mathit{filter}\ \mathit{even}\ \mathit{xs}) \\
&\mathit{sumOdds}' \ \ = \mathit{sum}\ (\mathit{filter}\ \mathit{odd}\ \mathit{xs})
\end{aligned}
$$

- Less code to write, and less code to debug.

Why are higher order functions useful?

- A higher-order function can do much more than a "first order" one, because a part of its behaviour can be controlled by the caller.
  - We can replace many similar functions by one higher-order function, parameterised on the differences.
- **Common programming idioms** can be encoded as functions within the language itself.
- **Domain specific languages** can be defined as collections of higher-order functions.
- **Algebraic properties** of higher-order functions can be used to reason about programs.

LAMBDA FUNCTIONS
Anonymous functions in Haskell

Lambda functions

The function definition in Haskell:

$$function\ param = expression$$

is internally represented by a *lambda function* in the compiler

$$function = \lambda param \rightarrow expression$$

Above, the $(\lambda param \rightarrow expression)$ is a lambda function. It is Haskell's way of defining *anonymous* functions, i.e. functions without a name.
When programming, we write `(\param -> expression)`.

Lambda functions - Multiple parameters

Several equivalent ways of defining the same function:

$$
\begin{aligned}
&f :: Num\ a \Rightarrow a \rightarrow a \rightarrow a \\
&f\ x\ y = 2 * x - y \\
&f\ x \quad = \lambda y \rightarrow 2 * x - y \\
&f \qquad = \lambda x \rightarrow \lambda y \rightarrow 2 * x - y \\
&f \qquad = \lambda x \rightarrow (\lambda y \rightarrow 2 * x - y) \\
&f \qquad = \lambda x\ y \rightarrow 2 * x - y
\end{aligned}
$$

Pattern matching on tuples is easily done using lambda functions also:

$$
\begin{aligned}
&g :: Num\ a \rightarrow (a, a) \rightarrow a \\
&g\ (x, y) = 2 * x - y \\
&g \qquad\quad = \lambda(x, y) \rightarrow 2 * x - y
\end{aligned}
$$

Lambda functions - Pattern matching

Consider a function using pattern matching

$$\begin{aligned} &sum\ [\,] &&= 0 \\ &sum\ (x : xs) &&= x + sum\ xs \end{aligned}$$

As a lambda-function, we would define it by

$$\begin{aligned} &sum = \lambda a \rightarrow \textbf{case}\ a\ \textbf{of} \\ &\quad [\,] \qquad\quad \rightarrow 0 \\ &\quad (x : xs) \rightarrow x + sum\ xs \end{aligned}$$

This is also what the compiler does internally

Lambda functions - Guards

If you want guards

$$\begin{aligned} sign\ n \mid n < 0 &= -1 \\ \mid n \equiv 0 &= 0 \\ \mid otherwise &= 1 \end{aligned}$$

but also want to write it in a lambda function, you can write

$$\begin{aligned} &sign = \lambda n \rightarrow \textbf{case}\ ()\ \textbf{of} \\ &\quad \_ \mid n < 0 \rightarrow -1 \\ &\quad \_ \mid n \equiv 0 \rightarrow 0 \\ &\quad \_ \qquad\qquad \rightarrow 1 \end{aligned}$$

Lambda functions - Local variables

If you want local variables

$$
\begin{aligned}
&\mathit{roots}\ a\ b\ c \mid d < 0 \ = [\,] \\
&\qquad\qquad\quad\ \mid d \equiv 0 = [-b \mathbin{/} (2 * a)] \\
&\qquad\qquad\quad\ \mid d > 0 \ = [(-b + \mathit{sqrt}\ d) \mathbin{/} (2 * a), (-b - \mathit{sqrt}\ d) \mathbin{/} (2 * a)] \\
&\quad \textbf{where} \\
&\qquad d = b \uparrow 2 - 4 * a * c
\end{aligned}
$$

you can use **let**-statements:

$$
\begin{aligned}
\mathit{roots} = \lambda a\ b\ c \rightarrow\ & \textbf{let}\ d = b \uparrow 2 - 4 * a * c \\
& \textbf{in case}\ (\,)\ \textbf{of} \\
& \quad \_ \mid d < 0 \ = [\,] \\
& \quad \_ \mid d \equiv 0 = [-b \mathbin{/} (2 * a)] \\
& \quad \_ \mid d > 0 \ = [(-b + \mathit{sqrt}\ d) \mathbin{/} (2 * a), \ldots]
\end{aligned}
$$

map, *filter* and *concat*

$$
\begin{aligned}
&\mathit{map} :: (a \to b) \to ([a] \to [b]) \\
&\mathit{map}\ f\ [\,] && = [\,] \\
&\mathit{map}\ f\ (x : xs) && = f\ x : \mathit{map}\ f\ xs \\
&\mathit{map}\ f\ xs && = [f\ x \mid x \leftarrow xs]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{filter} :: (a \to \mathit{Bool}) \to ([a] \to [a]) \\
&\mathit{filter}\ p\ [\,] && = [\,] \\
&\mathit{filter}\ p\ (x : xs) \mid p\ x && = x : \mathit{filter}\ p\ xs \\
&\qquad\qquad\qquad\ \mid \mathit{otherwise} && = \mathit{filter}\ p\ xs \\
&\mathit{filter}\ p\ xs && = [x \mid x \leftarrow xs, p\ x]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{concat} :: [[a]] \to [a] \\
&\mathit{concat}\ [\,] && = [\,] \\
&\mathit{concat}\ (xs : xss) && = xs +\!\!+\ \mathit{concat}\ xss \\
&\mathit{concat}\ xss && = [x \mid xs \leftarrow xss, x \leftarrow xs]
\end{aligned}
$$

Equational reasoning: Identities for *map*

$$
\begin{aligned}
map\ id\ xs &\equiv xs \\
map\ f\ (map\ g\ xs) &\equiv map\ (f \circ g)\ xs
\end{aligned}
$$

where

$$
\begin{aligned}
&id :: a \to a \\
&id\ x = x
\end{aligned}
$$

and

$$
\begin{aligned}
&(\circ) :: (b \to c) \to (a \to b) \to (a \to c) \\
&(f \circ g)\ x = f\ (g\ x)
\end{aligned}
$$

map, *filter* and *concat*: The basic primitives of list comprehensions

Just 3 functions makes it possible to express every program written using list comprehensions

$$
\begin{aligned}
[f\ x \mid x \leftarrow xs] &\equiv map\ f\ xs \\
[x \mid x \leftarrow xs, p\ x] &\equiv filter\ p\ xs \\
[e \mid Q, P] &\equiv concat\ [[e \mid P] \mid Q] \\
[e \mid Q, x \leftarrow [d \mid P]] &\equiv [e\ [x := d] \mid Q, P]
\end{aligned}
$$

EXAMPLE 1

$$
\begin{aligned}
& [x \mid xs \leftarrow xss, \neg\ (null\ xs), x \leftarrow xs] \\
\equiv\ & concat\ [[x \mid x \leftarrow xs] \mid xs \leftarrow xss, \neg\ (null\ xs)] \\
\equiv\ & concat\ [xs \mid xs \leftarrow xss, \neg\ (null\ xs)] \\
\equiv\ & concat\ (filter\ (\neg \circ null)\ xss)
\end{aligned}
$$

map, *filter* and *concat*: The basic primitives of list comprehensions

Just 3 functions makes it possible to express every program written using list comprehensions

$$
\begin{aligned}
[f\ x \mid x \leftarrow xs] &\equiv \mathit{map}\ f\ xs \\
[x \mid x \leftarrow xs, p\ x] &\equiv \mathit{filter}\ p\ xs \\
[e \mid Q, P] &\equiv \mathit{concat}\ [[e \mid P] \mid Q] \\
[e \mid Q, x \leftarrow [d \mid P]] &\equiv [e\ [x := d] \mid Q, P]
\end{aligned}
$$

EXAMPLE 2

$$
\begin{aligned}
& [y \mid xs \leftarrow xss, y \leftarrow [x * x \mid x \leftarrow xs]] \\
\equiv\ & [x * x \mid xs \leftarrow xss, x \leftarrow xs] \\
\equiv\ & \mathit{concat}\ [[x * x \mid x \leftarrow xs] \mid xs \leftarrow xss] \\
\equiv\ & \mathit{concat}\ [\mathit{map}\ \mathit{square}\ xs \mid xs \leftarrow xss] \\
\equiv\ & \mathit{concat}\ (\mathit{map}\ (\mathit{map}\ \mathit{square})\ xss)
\end{aligned}
$$

Function composition and function application

$$
\begin{aligned}
&(\circ) :: (b \to c) \to (a \to b) \to (a \to c)\\
&(f \circ g)\ x = f\ (g\ x)\\
&(\$) :: (a \to b) \to a \to b\\
&f\ \$\ x \qquad = f\ x
\end{aligned}
$$

Identities for *map*

$$
\begin{aligned}
map\ (f \circ g) &\equiv map\ f \circ map\ g \\
map\ f \circ concat &\equiv concat \circ map\ (map\ f) \\
map\ f\ (xs +\!\!+ ys) &\equiv map\ f\ xs +\!\!+ map\ f\ ys
\end{aligned}
$$

$$
\begin{aligned}
map &:: (a \to b) \to [a] \to [b] \\
map \circ map &:: (a \to b) \to [[a]] \to [[b]] \\
map \circ map \circ map &:: (a \to b) \to [[[a]]] \to [[[b]]]
\end{aligned}
$$

Identities for *filter*

$$
\begin{aligned}
\mathit{filter}\ p \circ \mathit{filter}\ q &\equiv \mathit{filter}\ q \circ \mathit{filter}\ p \\
\mathit{filter}\ p \circ \mathit{filter}\ q &\equiv \mathit{filter}\ (\lambda x \rightarrow p\ x \wedge q\ x) \\
\mathit{filter}\ p \circ \mathit{concat} &\equiv \mathit{concat} \circ \mathit{map}\ (\mathit{filter}\ p) \\
\mathit{filter}\ p\ (\mathit{xs} +\!\!+\ \mathit{ys}) &\equiv (\mathit{filter}\ p\ \mathit{xs}) +\!\!+ (\mathit{filter}\ p\ \mathit{ys})
\end{aligned}
$$

The constant function *const*

$$const :: a \rightarrow b \rightarrow a$$
$$const\ x = \backslash\_ \rightarrow x$$

Sometimes useful in combination with higher-order functions.
The length-function could be defined by

$$length :: [a] \rightarrow Int$$
$$length\ xs = sum\ (map\ (const\ 1)\ xs)$$

$$length' :: [a] \rightarrow Int$$
$$length' = sum \circ map\ (const\ 1)$$

The identity function *id*

$$id :: a \to a$$
$$id\ x = x$$

Sometimes useful in combination with higher-order functions.

$$or :: [Bool] \to Bool$$
$$or = \neg \circ null \circ filter\ id$$

$$\begin{aligned} or\ [True, False, False] &= (\neg \circ (null \circ filter\ id))\ [True, False, False] \\ &= \neg\ ((null \circ filter\ id)\ [True, False, False]) \\ &= \neg\ (null\ (filter\ id\ [True, False, False])) \\ &= \neg\ (null\ [True]) \\ &= \neg\ False \\ &= True \end{aligned}$$

How would we define *and*? $and = null \circ filter\ \neg$

curry, uncurry, flip

- *curry* converts a function on pairs to a curried function.
- *uncurry* converts a curried function to a function on pairs.
- *flip* flips the arguments of a function

$$
\begin{aligned}
&curry && :: ((a, b) \to c) \to (a \to b \to c) \\
&curry\ f\ x\ y && = f\ (x, y) \\
&uncurry && :: (a \to b \to c) \to ((a, b) \to c) \\
&uncurry\ f\ (x, y) && = f\ x\ y \\
&flip && :: (b \to a \to c) \to (a \to b \to c) \\
&flip\ f\ a\ b && = f\ b\ a
\end{aligned}
$$

We could have defined

$$zipWith\ f\ xs\ ys = map\ (uncurry\ f)\ (zip\ xs\ ys)$$

Puzzle for the break

$$
\begin{aligned}
&id && :: a \to a \\
&id\ x && = x \\
&flip && :: (b \to a \to c) \to (a \to b \to c) \\
&flip\ f\ a\ b && = f\ b\ a \\
&curry && :: ((a, b) \to c) \to (a \to b \to c) \\
&curry\ f\ x\ y && = f\ (x, y) \\
&uncurry && :: (a \to b \to c) \to ((a, b) \to c) \\
&uncurry\ f\ (x, y) && = f\ x\ y
\end{aligned}
$$

Use only the functions above to define a function

$$
\begin{aligned}
&swap :: (a, b) \to (b, a) \\
&swap = ???
\end{aligned}
$$

Solution: $(uncurry\ (flip\ (curry\ id)))$

FOLD AND UNFOLD

Abstract away recursion

$$\begin{aligned} [\,] +\!\!+\, ys &= ys \\ (x : xs) +\!\!+\, ys &= x : (xs +\!\!+\, ys) \end{aligned}$$

$$\begin{aligned} & (x1 : (x2 : (x3 : [\,]))) +\!\!+\, ys \\ \equiv\ & x1 : (x2 : (x3 : ys)) \end{aligned}$$

$$\begin{aligned} \mathit{concat}\ [\,] &= [\,] \\ \mathit{concat}\ (xs : xss) &= xs +\!\!+\, \mathit{concat}\ xss \end{aligned}$$

$$\begin{aligned} & \mathit{concat}\ (x1 : (x2 : (x3 : [\,]))) \\ \equiv\ & x1 +\!\!+\, (x2 +\!\!+\, (x3 +\!\!+\, [\,])) \end{aligned}$$

$$\begin{aligned} \mathit{and}\ [\,] &= \mathit{True} \\ \mathit{and}\ (x : xs) &= x \wedge \mathit{and}\ xs \end{aligned}$$

$$\begin{aligned} & \mathit{and}\ (x1 : (x2 : (x3 : [\,]))) \\ =\ & x1 \wedge (x2 \wedge (x3 \wedge \mathit{True})) \end{aligned}$$

$$\begin{aligned} \mathit{length}\ [\,] &= 0 \\ \mathit{length}\ (x : xs) &= 1 + \mathit{length}\ xs \end{aligned}$$

$$\begin{aligned} & \mathit{length}\ (x1 : (x2 : (x3 : [\,]))) \\ \equiv\ & 1 + (1 + (1 + [\,])) \end{aligned}$$

$$\begin{aligned} \mathit{reverse}\ [\,] &= [\,] \\ \mathit{reverse}\ (x : xs) &= \mathit{reverse}\ xs +\!\!+\, [x] \end{aligned}$$

$$\begin{aligned} & \mathit{reverse}\ (x1 : (x2 : (x3 : [\,]))) \\ \equiv\ & (([\,] +\!\!+\, [x3]) +\!\!+\, [x2]) +\!\!+\, [x1] \end{aligned}$$

Abstract away recursion

$$
\begin{aligned}
[\,] +\!\!+\, ys &= ys \\
(x : xs) +\!\!+\, ys &= x : (xs +\!\!+\, ys)
\end{aligned}
$$

$$
\begin{aligned}
concat\ [\,] &= [\,] \\
concat\ (xs : xss) &= xs +\!\!+\, concat\ xss
\end{aligned}
$$

$$
\begin{aligned}
and\ [\,] &= True \\
and\ (x : xs) &= x \wedge and\ xs
\end{aligned}
$$

$$
\begin{aligned}
length\ [\,] &= 0 \\
length\ (x : xs) &= 1 + length\ xs
\end{aligned}
$$

$$
\begin{aligned}
reverse\ [\,] &= [\,] \\
reverse\ (x : xs) &= reverse\ xs +\!\!+\, [x]
\end{aligned}
$$

In general, all of the functions mentioned here are of the form

$$
\begin{aligned}
h\ [\,] &= e \\
h\ (x : xs) &= x \oplus h\ xs
\end{aligned}
$$

where e is an expression and $\oplus$ is some binary operator.
The result directly follows the structure of the list:

$$
\begin{aligned}
h\ \ &(x1 : (x2 : (x3 : [\,]))) \\
\equiv\ &(x1 \oplus (x2 \oplus (x3 \oplus e)))
\end{aligned}
$$

Abstract away recursion

$$
\begin{aligned}
[\,] +\!\!+\, ys &= ys \\
(x : xs) +\!\!+\, ys &= x : (xs +\!\!+\, ys)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{concat}\ [\,] &= [\,] \\
\mathit{concat}\ (xs : xss) &= xs +\!\!+\, \mathit{concat}\ xss
\end{aligned}
$$

$$
\begin{aligned}
\mathit{and}\ [\,] &= \mathit{True} \\
\mathit{and}\ (x : xs) &= x \wedge \mathit{and}\ xs
\end{aligned}
$$

$$
\begin{aligned}
\mathit{length}\ [\,] &= 0 \\
\mathit{length}\ (x : xs) &= 1 + \mathit{length}\ xs
\end{aligned}
$$

$$
\begin{aligned}
\mathit{reverse}\ [\,] &= [\,] \\
\mathit{reverse}\ (x : xs) &= \mathit{reverse}\ xs +\!\!+\, [x]
\end{aligned}
$$

In general, all of the functions mentioned here are of the form

$$
\begin{aligned}
h\ [\,] &= e \\
h\ (x : xs) &= x \oplus h\ xs
\end{aligned}
$$

where e is an expression and $\oplus$ is some binary operator. What is e and $\oplus$ for each of the functions on the left?

- $(+\!\!+) :: [a] \to [a] \to [a]$
- $\mathit{concat} :: [[a]] \to [a]$
- $\mathit{and} :: [\mathit{Bool}] \to \mathit{Bool}$
- $\mathit{length} :: [a] \to \mathit{Int}$
- $\mathit{reverse} :: [a] \to [a]$

foldr to the rescue

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

$$
\mathit{foldr}\ (\oplus)\ e\ [x1, x2, x3] \equiv x1 \oplus (x2 \oplus (x3 \oplus e))
$$

The functions from before become:

$$
\begin{aligned}
xs +\!\!+\ ys &= \mathit{foldr}\ (:)\ ys\ xs \\
\mathit{concat}\ xss &= \mathit{foldr}\ (+\!\!+)\ [\,]\ xss \\
\mathit{and}\ xs &= \mathit{foldr}\ (\wedge)\ \mathit{True}\ xs \\
\mathit{length}\ xs &= \mathit{foldr}\ (\mathit{const}\ (1+))\ 0\ xs \\
\mathit{reverse}\ xs &= \mathit{foldr}\ (\lambda x\ ys \to ys +\!\!+ [x])\ [\,]\ xs
\end{aligned}
$$

Fixing *reverse*

Recall, that we last time defined

$$
\begin{aligned}
&\mathit{reverse} :: [a] \to [a]\\
&\mathit{reverse}\ [\,] \qquad\quad = [\,]\\
&\mathit{reverse}\ (x : xs) = \mathit{reverse}\ xs \mathbin{+\!\!+} [x]
\end{aligned}
$$

and that it used $O(n^2)$ reductions, due to the fact that there are n applications of $(+\!\!+)$, where the i'th application consists of i reductions.

Fixing *reverse*

Now we propose a new solution which avoids $(+\!\!+)$:

$$
\begin{aligned}
&\mathit{reverse} :: [a] \to [a] \\
&\mathit{reverse}\ \mathit{xs} = \mathit{reverse}'\ \mathit{xs}\ [\,] \\
&\quad \textbf{where} \\
&\qquad \mathit{reverse}'\ [\,]\ \mathit{ys} = \mathit{ys} \\
&\qquad \mathit{reverse}'\ (x : \mathit{xs})\ \mathit{ys} = \mathit{reverse}'\ \mathit{xs}\ (x : \mathit{ys})
\end{aligned}
$$

$$
\begin{aligned}
\mathit{reverse}\ [1,2,3] &\equiv \mathit{reverse}'\ [1,2,3]\ [\,] \\
&\equiv \mathit{reverse}'\ [2,3]\ [1] \\
&\equiv \mathit{reverse}'\ [3]\ [2,1] \\
&\equiv \mathit{reverse}'\ [\,]\ [3,2,1] \\
&\equiv [3,2,1]
\end{aligned}
$$

$O(n)$ reductions!

Introducing *foldl*

We want to generalize the concept of recursive function on lists, which use an accumulating parameter. This is all functions which have the following structure:

$$\begin{aligned} h\ [\,]\ acc &= acc \\ h\ (x : xs)\ acc &= h\ xs\ (acc \oplus x) \end{aligned}$$

for some some binary operation $\oplus$ and a value e. Execution results in evaluation:

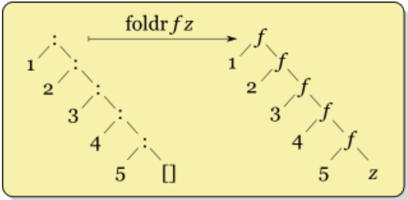
$$h\ [x1, x2, x3]\ e \equiv ((e \oplus x1) \oplus x2) \oplus x3$$

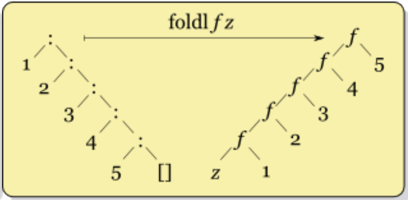
Introducing *foldl*

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

$$foldl :: (a \to b \to a) \to a \to [b] \to a$$
$$foldl\ f\ z\ [\,] = z$$
$$foldl\ f\ z\ (x : xs) = foldl\ f\ (f\ z\ x)\ xs$$

$$foldr\ (\oplus)\ e\ [x1, x2, x3] \equiv x1 \oplus (x2 \oplus (x3 \oplus e))$$
$$foldl\ (\oplus)\ e\ [x1, x2, x3] \equiv ((e \oplus x1) \oplus x2) \oplus x3$$





Examples

$$
\begin{aligned}
\mathit{reverse}\ \mathit{xs} &= \mathit{foldl}\ (\mathit{flip}\ (:))\ [\,]\ \mathit{xs}\\
\mathit{sum}\ \mathit{xs} &= \mathit{foldl}\ (+)\ 0\ \mathit{xs}\\
\mathit{product}\ \mathit{xs} &= \mathit{foldl}\ (*)\ 1\ \mathit{xs}\\
\mathit{maximum}\ [\,] &= \mathit{error}\ \texttt{"Empty list"}\\
\mathit{maximum}\ (x : \mathit{xs}) &= \mathit{foldl}\ \mathit{max}\ x\ \mathit{xs}
\end{aligned}
$$

Functions using folds and their time complexities

	f	z	$foldl$	$foldr$
$reverse\ xs$	$flip\ (:) :: [a] \to a \to [a]$	$[\,]$	$O(n)$	-
$reverse\ xs$	$(\lambda x\ z \to z +\!\!+ [x]) :: a \to [a] \to [a]$	$[\,]$	-	$O(n^2)$
$sum\ xs$	$(+) :: Num\ a \Rightarrow a \to a \to a$	0	$O(n)$	$O(n)$
$product\ xs$	$(*) :: Num\ a \Rightarrow a \to a \to a$	1	$O(n)$	$O(n)$
$maximum\ (x : xs)$	$max :: Ord\ a \Rightarrow a \to a \to a$	x	$O(n)$	$O(n)$
$minimum\ (x : xs)$	$min :: Ord\ a \Rightarrow a \to a \to a$	x	$O(n)$	$O(n)$
$xs +\!\!+ ys$	$(:) :: a \to [a] \to [a]$	ys	-	$O(n)$
$concat\ xss$	$(+\!\!+) :: [a] \to [a] \to [a]$	$[\,]$	$O(m^2 n)$	$O(mn)$[1]
$length\ xs$	$const\ (1+) :: a \to Int \to Int$	0	-	$O(n)$
$and\ xs$	$(\wedge) :: Bool \to Bool \to Bool$	$True$	$O(n)$	$O(i)$[2]
$or\ xs$	$(\vee) :: Bool \to Bool \to Bool$	$False$	$O(n)$	$O(i)$[3]

[1] m is length of xss, n is length of sublists
[2] i is index of first *False* occurence
[3] i is index of first *True* occurence

Elaboration on *concat*

$$
\begin{aligned}
\mathit{concat}\ [x1, x2 \ldots xn] &\equiv \mathit{foldl}\ (+\!\!+)\ []\ [x1, x2 \ldots xn] \\
&\equiv (\ldots([] +\!\!+ x1) +\!\!+ x2) +\!\!+ \ldots) +\!\!+ xn
\end{aligned}
$$

$(+\!\!+)$ runs over all preceding lists when appending a new list!
$\rightarrow O(m^2 n)$.

$$
\begin{aligned}
\mathit{concat}\ [x1, x2 \ldots xn] &\equiv \mathit{foldr}\ (+\!\!+)\ []\ [x1, x2 \ldots xn] \\
&\equiv x1 +\!\!+ (x2 +\!\!+ (\ldots(xn +\!\!+ [])))
\end{aligned}
$$

This one only runs through each sublist once. $\rightarrow O(mn)$.

First duality theorem

Suppose $\oplus$ is associative with unit e. Then

$$foldr\ (\oplus)\ e\ xs = foldl\ (\oplus)\ e\ xs$$

for all finite lists xs. **Examples:**

$$\begin{aligned} sum &= foldl\ (+)\ 0 \\ and &= foldl\ (\wedge)\ True \\ concat &= foldl\ (+\!\!+)\ [\,] \end{aligned}$$

So: In this case, *foldr* and *foldl* gives the same result, but their performance characteristics might differ!
A type with such operations $+$ and e is called a **Monoid**, and in Haskell, there is a corresponding type class.

How would *map* look like for *Tree a*?

Recall that for Lists, we write a *map* function like this:

$$\textbf{data}\ List\ a = Nil \mid Cons\ a\ (List\ a)$$

$$\begin{aligned}
&listMap :: (a \to b) \to List\ a \to List\ b\\
&listMap\ f\ Nil \qquad\qquad\quad = Nil\\
&listMap\ f\ (Cons\ x\ xs) = Cons\ (f\ x)\ (listMap\ f\ xs)
\end{aligned}$$

$$\textbf{data}\ Tree\ a = Nil \mid Node\ a\ (Tree\ a)\ (Tree\ a)$$

$$\begin{aligned}
&treeMap :: (a \to b) \to Tree\ a \to Tree\ b\\
&treeMap\ f\ Nil \qquad\qquad\quad = Nil\\
&treeMap\ f\ (Node\ x\ l\ r) = Node\ (f\ x)\ (treeMap\ f\ l)\ (treeMap\ f\ r)
\end{aligned}$$

$$\begin{aligned}
&\textbf{instance}\ (Functor\ Tree)\ \textbf{where}\\
&\quad fmap\ f\ Nil \qquad\qquad\quad = Nil\\
&\quad fmap\ f\ (Node\ x\ l\ r) = Node\ (f\ x)\ (fmap\ f\ l)\ (fmap\ f\ r)
\end{aligned}$$

The Functor Type Class

$$
\begin{array}{l}
\textbf{class}\ \mathit{Functor}\ f\ \textbf{where} \\
\quad \mathit{fmap} :: (a \to b) \to f\ a \to f\ b
\end{array}
$$

Note that f in the above should be a *type constructor*.

Every functor F should satisfy the Functor laws:

$$
\begin{aligned}
\mathit{fmap}\ \mathit{id} &\equiv \mathit{id} \\
\mathit{fmap}\ (f \circ g) &\equiv \mathit{fmap}\ f \circ \mathit{fmap}\ g
\end{aligned}
$$

More examples:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Functor}\ ((,)\ a)\ \textbf{where} \\
\quad \mathit{fmap}\ f\ (x, y) = (x, f\ y)
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Functor}\ (\mathit{Maybe})\ \textbf{where} \\
\quad \mathit{fmap}\ f\ \mathit{Nothing} = \mathit{Nothing} \\
\quad \mathit{fmap}\ f\ (\mathit{Just}\ x) = \mathit{Just}\ (f\ x)
\end{array}
$$

Unfolding

$$unfoldr :: (b \to Maybe\ (a, b)) \to b \to [a]$$

The *unfoldr* function is a 'dual' to *foldr*: While *foldr* reduces a list to a summary value, *unfoldr* builds a list from a seed value. The function takes the element and

- returns *Nothing* if it is done producing the list or
- returns $Just\ (a, b)$, in which case, a is a prepended to the list and b is used as the next element in a recursive call.

unfoldr example: *digits*

$$unfoldr :: (b \to Maybe\ (a, b)) \to b \to [a]$$

In the first lecture, we saw

$$
\begin{aligned}
&digits :: Integer \to [Integer] \\
&digits\ n \\
&\quad \mid n \equiv 0 \qquad = [\,] \\
&\quad \mid otherwise = n \text{ `}mod\text{` } 10 : digits\ (n \text{ `}div\text{` } 10)
\end{aligned}
$$

This function corresponds to an “unfoldr” of the number, namely

$$
\begin{aligned}
&digits = unfoldr\ (\lambda n \to \mathbf{case}\ n\ \mathbf{of} \\
&\quad 0 \to Nothing \\
&\quad \_ \to Just\ (n \text{ `}mod\text{` } 10, n \text{ `}div\text{` } 10) \\
&)
\end{aligned}
$$

unfoldr example: *digits*

Syddansk Universitet
University of Southern Denmark

$$unfoldr :: (b \rightarrow Maybe\ (a, b)) \rightarrow b \rightarrow [a]$$

$$
\begin{array}{l}
digits :: Integer \rightarrow [Integer] \\
digits = unfoldr\ (\lambda n \rightarrow \mathbf{case}\ n\ \mathbf{of} \\
\quad 0 \rightarrow Nothing \\
\quad \_ \rightarrow Just\ (n\ `mod`\ 10, n\ `div`\ 10) \\
)
\end{array}
$$

In this case, we can fold/unfold back and forth:

$$
\begin{array}{ll}
digits\ 1337 & \equiv [7, 3, 3, 1] \\
foldr\ (\lambda x\ y \rightarrow x + 10 * y)\ 0\ [7, 3, 3, 1] & \equiv 1337
\end{array}
$$

Discovering variants of the fold function

- Recall the type

$$foldr :: (a \to b \to b) \to b \to [a] \to b$$

- We could have given *foldr* a similar type to *unfoldr*.
  To give it a different name, the fold function is here named *cata* (short for *catamorphism* - the general name for folds over algebraic data types)

$$\begin{aligned} unfoldr &:: (b \to Maybe\ (a, b)) \to (b \to [a]) \\ cata &:: (Maybe\ (a, b) \to b) \to ([a] \to b) \end{aligned}$$

  where e.g. *foldr* $(+)$ 0 would correspond to *cata plus* with

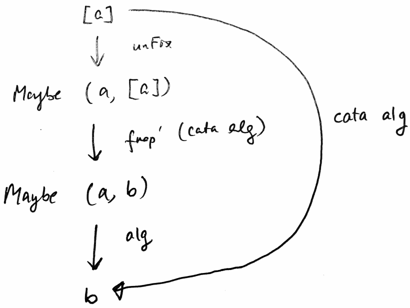
$$\begin{aligned} &plus\ Nothing = 0 \\ &plus\ (Just\ (x, y)) = x + y \end{aligned}$$

Discovering variants of the fold function

$$
\begin{aligned}
&\mathit{unFix} &&:: [a] \to \mathit{Maybe}\ (a,[a]) \\
&\mathit{unFix}\ [\,] &&= \mathit{Nothing} \\
&\mathit{unFix}\ (x:xs) &&= \mathit{Just}\ (x,xs)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{fmap}' :: (b \to b') \to (\mathit{Maybe}\ (a,b) \to \mathit{Maybe}\ (a,b')) \\
&\mathit{fmap}' = \mathit{fmap} \circ \mathit{fmap}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{cata} &&:: (\mathit{Maybe}\ (a,b) \to b) \to ([a] \to b) \\
&\mathit{cata}\ \mathit{alg} = &&\mathit{alg} \circ \mathit{fmap}'\ (\mathit{cata}\ \mathit{alg}) \circ \mathit{unFix}
\end{aligned}
$$



Discovering variants of the fold function

In general, for a recursive algebraic datatype - e.g.

$$\textbf{data}\ List\ a = Nil \mid Cons\ a\ (List\ a)$$

we can associate a non-recursive variant, by introducing an extra type parameter:

$$\textbf{data}\ ListF\ a\ b = NilF \mid ConsF\ a\ b$$

The fold-function would then be of the type

$$cata :: (ListF\ a\ b \to b) \to (List\ a \to b)$$

The types $ListF\ a\ b$ and $Maybe\ (a, b)$ are isomorphic.

- *Nothing* corresponds to *NilF*
- $Just\ (x, y)$ corresponds to $ConsF\ x\ y$

Complete example of "general" fold on lists

$$\textbf{data}\ \mathit{ListF}\ a\ b = \mathit{NilF}\ \mid \mathit{ConsF}\ a\ b$$

$$\textbf{data}\ \mathit{List}\ a = \mathit{Nil}\ \mid \mathit{Cons}\ a\ (\mathit{List}\ a)$$

$$\begin{aligned}
&\mathit{unFix} :: \mathit{List}\ a \to \mathit{ListF}\ a\ (\mathit{List}\ a)\\
&\mathit{unFix}\ \mathit{Nil} = \mathit{NilF}\\
&\mathit{unFix}\ (\mathit{Cons}\ x\ xs) = \mathit{ConsF}\ x\ xs
\end{aligned}$$

$$\begin{aligned}
&\textbf{instance}\ \mathit{Functor}\ (\mathit{ListF}\ a)\ \textbf{where}\\
&\quad \mathit{fmap}\ _\ \mathit{NilF} = \mathit{NilF}\\
&\quad \mathit{fmap}\ f\ (\mathit{ConsF}\ x\ xs) = \mathit{ConsF}\ x\ (f\ xs)
\end{aligned}$$

$$\begin{aligned}
&\mathit{cata} :: (\mathit{ListF}\ a\ b \to b) \to (\mathit{List}\ a \to b)\\
&\mathit{cata}\ \mathit{alg} = \mathit{alg} \circ \mathit{fmap}\ (\mathit{cata}\ \mathit{alg}) \circ \mathit{unFix}
\end{aligned}$$

Final example

$$
\begin{aligned}
&\textbf{data}\ \mathit{ListF}\ a\ b \quad = \mathit{NilF} \mid \mathit{ConsF}\ a\ b \\
&\textbf{newtype}\ \mathit{Fix}\ f = \mathit{Fx}\ (f\ (\mathit{Fix}\ f)) \\
&\textbf{type}\ \mathit{List}\ a \qquad = \mathit{Fix}\ (\mathit{ListF}\ a)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{unFix} :: \mathit{Fix}\ f \to f\ (\mathit{Fix}\ f) \\
&\mathit{unFix}\ (\mathit{Fx}\ x) = x
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Functor}\ (\mathit{ListF}\ a)\ \textbf{where} \\
&\quad \mathit{fmap}\ \_\ \mathit{NilF} \qquad\qquad\quad = \mathit{NilF} \\
&\quad \mathit{fmap}\ f\ (\mathit{ConsF}\ x\ \mathit{xs}) = \mathit{ConsF}\ x\ (f\ \mathit{xs})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{cata} :: (\mathit{ListF}\ a\ b \to b) \to (\mathit{List}\ a \to b) \\
&\mathit{cata}\ \mathit{alg} = \mathit{alg} \circ \mathit{fmap}\ (\mathit{cata}\ \mathit{alg}) \circ \mathit{unFix}
\end{aligned}
$$