A start on higher types: Mapping, 1

Mapping via list comprehension

\[
\begin{align*}
\text{doubleAll} & : [\text{Int}] \rightarrow [\text{Int}] \\
\text{doubleAll} \hspace{1pt} \text{lst} & = [2\times \mid x \leftarrow \text{lst}] \\
\text{addPairs} & : [(\text{Int},\text{Int})] \rightarrow [[\text{Int}]] \\
\text{addPairs} \hspace{1pt} \text{mns} & = [[m+n] \mid (m,n) \leftarrow \text{mns}] \\
\text{multAll} & : \text{Int} \rightarrow [\text{Int}] \rightarrow [\text{Int}] \\
\text{multAll} \hspace{1pt} x \hspace{1pt} \text{ys} & = [x\cdot y \mid y \leftarrow \text{ys}] \\
\end{align*}
\]

More generally for any function \( f : a \rightarrow b \), we can define a function

\[
\begin{align*}
\text{apply}_f & : [a] \rightarrow [b] \\
\text{apply}_f \hspace{1pt} \text{xs} & = [f \hspace{1pt} x \mid x \leftarrow \text{xs}] \\
\end{align*}
\]
A start on higher types: Mapping, 3

Mapping via \texttt{map}

Let us define a \textit{generic} function to do mapping:

\begin{verbatim}
map :: (a -> b) -> [a] -> [b]
map f lst = [ f x | x <- lst ]

map' :: (a -> b) -> [a] -> [b]
map' f [] = []
map' f (x:xs) = (f x):map' f xs
\end{verbatim}

\texttt{map} is higher order, it accepts a function as an argument. E.g.,

\begin{verbatim}
map fst [(1,False), (3,True), (-5,False), (34,False)] ~ [1,3,-5,34]
map length [[1,5,6], [3,5], [], [3..10]] ~ [3,2,0,8]
map sum [[1,5,6], [3,5], [], [3..10]] ~ [12,8,0,52]
\end{verbatim}

A start on higher types: Filtering, 1

Filtering elements from a list via list comprehensions

\begin{verbatim}
lessThan10 :: [Int] -> [Int]
lessThan10 xs = [ x | x <- xs, x<10 ]

offDiagonal :: [(Int,Int)] -> [(Int,Int)]
offDiagonal mns = [(m,n) | (m,n) <- mns , m/=n]
\end{verbatim}

A start on higher types: Filtering, 2

Here is a generic way of doing filtering:

\begin{verbatim}
filter :: (a -> Bool) -> [a] -> [a]
filter p lst = [ x | x <- lst, p x ]

filter' :: (a -> Bool) -> [a] -> [a]
filter' p [] = []
filter' p (x:xs) | p x = x:(filter' p xs)
| otherwise = filter' p xs
\end{verbatim}

\texttt{filter} is higher order, it accepts functions as an argument. E.g.,

\begin{verbatim}
isOffDiag :: (Int,Int) -> Bool
isOffDiag (m,n) = (m/=n)

filter isOffDiag [(3,4),(5,5),(10,-2),(99,99)] ~ [(3,4),(10,-2)]
filter isDigit "a37b297" ~ "379?"
filter not [True,False,False,True] ~ [False,False]
\end{verbatim}

Functions as First-Class Values

In functional languages (generally), functions are \textit{first-class values}, i.e. are treated just like any other value.

So functions can be

- passed as arguments to functions
- returned as results from functions
- bound to variables
- expressed without being given a name (\texttt{\lambda}-expressions)
- elements of list (and other data structures)
- ...

A function that

(i) accepts functions as arguments or
(ii) returns a function as a value or
(iii) both (i) and (ii)

is higher order. E.g., \texttt{map} and \texttt{filter}.
Higher-type goodies, 1

**dropWhile, takeWhile**
:: (a -> Bool) -> [a] -> [a]
dropWhile p [] = []
dropWhile p (x:xs)
  | p x = dropWhile p xs
  | otherwise = x:xs
takeWhile p [] = []
takeWhile p (x:xs)
  | p x = x : takeWhile p xs
  | otherwise = []

**For example:**
takeWhile (<10) [0,3..20]  \(\sim\) \([0,3,6,9]\)
dropWhile (<10) [0,3..20]  \(\sim\) \([12,15,18]\)
dropWhile isSpace " hi there "  \(\sim\) " hi there "
takeWhile (not . isSpace) " hi there "  \(\sim\) " hi"
dropWhile (not . isSpace) " hi there "  \(\sim\) " there "

Q: What is (<10) doing?
Q: What is “.” doing??

Higher-type goodies, 2

**span**
:: (a -> Bool) -> [a] -> ([a],[a])
span p [] = ([],[])
span p xs@(x:xs')
  | p x = (x:ys,zs)
  | otherwise = ([],xs)
  where (ys,zs) = span p xs'

**For example:**
span (<10) [0,3..20]  \(\sim\) \(([0,3,6,9],[12,15,18])\)
span isSpace " hi there "  \(\sim\) "(",""hi there ")"

Q: What is the @ doing in “span p xs@(x:xs’)”?

Higher-type goodies, 3

**zipWith**
:: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith' _ [] [] = []
zipWith' _ _ [] = []
zipWith' f (x:xs) (y:ys) = f x y : zipWith' f xs ys

**For example:**
sum <$> zipWith (*) [2, 5, 3] [1.75, 3.45, 0.25]
  \(\sim\) \([3.5, 17.25, 0.75]\)
  \(\sim\) \(21.50\)
zipWith (\(\backslash \ a \ b \ -> \ (a * 30 + 3) / b\)) [5,4,3,2,1] [1,2,3,4,5]
  \(\sim\) \([153.0, 61.5, 31.0, 15.75, 6.6]\)

Q: What is the “$” doing??
Q: What is the (\(\backslash \ a \ b \ -> \ (a * 30 + 3) / b\)) doing?
Digression: The application operator

\( (\cdot) :: (a \to b) \to a \to b \)

\( f \cdot x = f \ x \quad -- \) has low, right-associative binding precedence

So

\[
\text{sum } \cdot \ (\text{filter (> 10)} \ (\text{map } (*2) \ [2..10])) \\
\equiv \text{sum } (\text{filter (> 10)} \ (\text{map } (*2) \ [2..10]))
\]

Digression: \( \lambda \)-expressions

The following definitions are equivalent

\[
\text{munge}, \text{munge'} :: \text{Int} \to \text{Int} \\
\text{munge } x = 3 \times x + 1 \\
\text{munge'} = \lambda x \to 3 \times x + 1
\]

So the following expressions are equivalent

\[
\text{map } \text{munge} \ [2..8] \\
\text{map } \text{munge'} \ [2..8] \\
\text{map } (\lambda x \to 3 \times x + 1) \ [2..8]
\]

So, \( (\lambda x \to 3 \times x + 1) \) defines a “nameless” function.

We can use \( (\lambda \rightarrow \text{Int}) \) to return functional results. E.g.,

\[
\text{addNum} :: \text{Int} \to (\text{Int} \to \text{Int}) \\
\text{addNum } n = \lambda x \to (x+n)
\]

Higher-types, structural recursion on lists, 1

Consider some structural recursion on lists:

\[
\begin{align*}
\text{sum'} [\ ] &= 0 \\
\text{sum'} (x:xs) &= x + \text{sum'} xs \quad -- = (+) x (\text{sum'} xs) \\
\text{concat'} [\ ] &= [] \\
\text{concat'} (xs:xss) &= xs ++ \text{concat'} xss \quad -- = (++) xs (\text{concat'} xss) \\
\text{unzip'} [\ ] &= ([],[]) \\
\text{unzip'} ((x,y):xys) &= (x:xs,y:ys) \quad -- \quad \text{where } f (a,b) (as,bs) = (a:as,b:bs)
\end{align*}
\]

These all have the general form:

\[
\begin{align*}
\text{someFun} [\ ] &= z \\
\text{someFun} (x:xs) &= f x (\text{someFun} xs)
\end{align*}
\]

So we can encapsulate this by:

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

Higher-types, structural recursion on lists, 2

As a foldr

\[
\begin{align*}
\text{sum'} [\ ] &= 0 \\
\text{sum'} (x:xs) &= x + \text{sum'} xs \\
\text{concat'} [\ ] &= [] \\
\text{concat'} (xs:xss) &= xs ++ \text{concat'} xss \\
\text{unzip'} [\ ] &= ([],[]) \\
\text{unzip'} ((x,y):xys) &= (x:xs,y:ys) \quad \text{where } f (x,y) (xs,ys) = (x:xs,y:ys)
\end{align*}
\]

Original

\[
\begin{align*}
\text{sum'} [\ ] &= 0 \\
\text{sum'} (x:xs) &= x + \text{sum'} xs \\
\text{concat'} [\ ] &= [] \\
\text{concat'} (xs:xss) &= xs ++ \text{concat'} xss \\
\text{unzip'} [\ ] &= ([],[]) \\
\text{unzip'} ((x,y):xys) &= (x:xs,y:ys) \quad \text{where } f (x,y) (xs,ys) = (x:xs,y:ys)
\end{align*}
\]
Higher-types, structural recursion on lists, 3

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)

Foldr’s cousin’s

Foldr’s cousin’s: alternatively, 1

For folds: look here
For scans: look here
Foldr’s cousin’s: alternatively, 1

Try:
- foldr1 max [1,4,8,4,9,4]
- foldl1 max [1,4,8,4,9,4]
- scanr1 max [1,4,8,4,9,4]
- scanl1 max [1,4,8,4,9,4]

Foldr’s cousin’s: alternatively, 2

Try:
- foldr1 max [1,4,8,4,9,4]
- foldl1 max [1,4,8,4,9,4]
- scanr1 max [1,4,8,4,9,4]
- scanl1 max [1,4,8,4,9,4]

For folds:
http://hackage.haskell.org/package/base-4.10.1.0/docs/Prelude.html#g:11

For scans:
http://hackage.haskell.org/package/base-4.10.1.0/docs/Prelude.html#g:16

Class Exercises

1. Use `foldr` to define \( n \mapsto 1^2 + 2^2 + 3^2 + \cdots + n^2 \).

2. Use `foldr` and `foldl` to define `length`.

3. Use `foldr` and `foldl` to define `and` and `or`.

4. Use `foldr` or `foldl` to define `reverse`.

5. Use `scanr` or `scanl` to define \( n \mapsto [1!, 2!, 3!, \ldots, n!] \).
Higher types

Class Exercises

1. Use \texttt{foldr} to define \( n \mapsto n^2 + 2 \).

2. Use \texttt{foldr} and \texttt{foldl} to define \texttt{length}.

3. Use \texttt{foldr} and \texttt{foldl} to define \texttt{and} and \texttt{or}.

4. Use \texttt{foldr} or \texttt{foldl} to define \texttt{reverse}.

5. Use \texttt{scanr} or \texttt{scanl} to define \( n \mapsto 1!, 2!, 3!, \ldots, n! \).

2018-01-30

Aside: Structural Recursions on Natural Numbers, 1

We can introduce a “natural number data type” by:

\[
data Nat = \text{Zero} | \text{Succ Nat}
\]

where \texttt{Zero} stands for 0 and \texttt{Succ} stands for the function \( x \mapsto x + 1 \).

A structural recursion over \texttt{Nat}’s is a function of the form:

\[
\text{fun} :: \text{Nat} \rightarrow a
\]

\[
\text{fun \ Zero} = z
\]

\[
\text{fun \ (Succ \ n)} = f \ (\text{fun \ n})
\]

where \( z :: a \) and \( f :: a \rightarrow a \). So if you expand things out, you see that

\[
\text{fun} \ (\text{Succ} \ (\text{Succ} \ (\ldots \text{Zero}))) = (f \ (f \ (\ldots \ z)))
\]

We can define a fold for \texttt{Nat}’s by:

\[
foldn :: (a\rightarrow a) \rightarrow a \rightarrow \text{Nat} \rightarrow a
\]

\[
foldn \ f \ z \ \text{Zero} = z
\]

\[
foldn \ f \ z \ (\text{Succ} \ n) = f \ (\text{foldn} \ f \ z \ n)
\]

Aside: Structural Recursions on Natural Numbers, 2

Using

\[
data Nat = \text{Zero} | \text{Succ Nat}
\]

\[
foldn :: (a\rightarrow a) \rightarrow a \rightarrow \text{Nat} \rightarrow a
\]

\[
foldn \ f \ z \ \text{Zero} = z
\]

\[
foldn \ f \ z \ (\text{Succ} \ n) = f \ (\text{foldn} \ f \ z \ n)
\]

we can bootstrap arithmetic by:

\[
\text{add} \ m \ n = \text{foldn} \ \text{Succ} \ n \ m
\]

\[
\text{times} \ m \ n = \text{foldn} \ (\text{‘add‘} \ n) \ \text{Zero} \ m
\]

Functions and types

In Haskell every function

\( \rightarrow \) takes exactly one argument and

\( \rightarrow \) returns exactly one value.

For example: \( f :: \text{Int} \rightarrow \text{Bool} \)

In general: \( g :: \text{t1} \rightarrow \text{t2} \)

Examples:

\( \rightarrow \) (\text{Int} \rightarrow \text{Bool} \rightarrow \text{Char}) \equiv \text{Int} \rightarrow \text{Bool} \rightarrow \text{Char}

\( \rightarrow \) associates to the right

\( t_1 \rightarrow t_2 \rightarrow \cdots \rightarrow t_n \rightarrow t \equiv t_1 \rightarrow (t_2 \rightarrow \cdots (t_n \rightarrow t) \cdots) \)
Associations

Convention: \( \rightarrow \) associates to the right

\[ t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow \cdots \rightarrow t_n \rightarrow t \equiv t_1 \rightarrow (t_2 \rightarrow (t_3 \rightarrow (\cdots (t_n \rightarrow t))\ldots)) \]

Convention: application associates to the left

\[ f x_1 x_2 x_3 \ldots x_n \equiv (\cdots ((f x_1)x_2)x_3) \ldots x_n) \]

WHY?

Suppose

\[
\begin{align*}
f &:: t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow t \\
e_1 &:: t_1 \\
e_2 &:: t_2 \\
e_3 &:: t_3
\end{align*}
\]

Then

\[
\begin{align*}
f e_1 &:: t_2 \rightarrow t_3 \rightarrow t \\
f e_1 e_2 &:: t_3 \rightarrow t \\
f e_1 e_2 e_3 &:: t
\end{align*}
\]

Currying and Uncurrying

Consider

\[
\begin{align*}
\text{comp1} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Bool} \\
\text{comp1} x y &= (x \leq y) \\
\text{comp2} :: (\text{Int}, \text{Int}) \rightarrow \text{Bool} \\
\text{comp2} (x, y) &= (x < y)
\end{align*}
\]

Every \( f :: t_1 \rightarrow t_2 \rightarrow \cdots \rightarrow t_n \rightarrow t \)
has a corresponding \( f' :: (t_1, t_2, \ldots, t_n) \rightarrow t \)
and vise versa.

In fact

\[
\begin{align*}
curry2 :: ((a, b) \rightarrow c) \rightarrow a \rightarrow b \rightarrow c \\
curry2 g &= \lambda x y \rightarrow g(x, y) \\
uncurry2 :: (a \rightarrow b \rightarrow c) \rightarrow (a, b) \rightarrow c \\
uncurry2 f &= \lambda (x, y) \rightarrow f x y
\end{align*}
\]

Mathematically: This is just a fancier version of:

\[
(c^b)^a = c^{a \times b}
\]

from High School math.

Was that so bad?