fixed typo
hamayumm authored 11 hours ago

1b83b650

higher-order_functions.md 16.9 KB

Higher-order Functions

These notes should be read in conjunction with chapter 7 - Higher-order functions of our textbook Programming in Haskell.

- We discuss some examples from the Haskell'98 standard prelude for pedagogical purposes.
- See the prelude for the current version of the language for all predefined classes and their instances.

Note:

Please ignore this declaration until you have read the *Base Conversion* section in the **Binary String Transmitter** example at the end of this handout. We mention it here, in order to ensure that the generated haskell file contains this import statement at the beginning.

```
import Data.Char
```

Basic Concepts

There is also a video on this section.

A function is called `higher-order` if it takes a function as an argument or returns a function as a result.

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)
```

For example:

```
> twice (*2) 3
12

> twice (+10) 5
25

> twice (\ x -> x ^ 2) 3
81

> twice reverse [1,2,3]
[1,2,3]
```

The function `twice` is higher-order because it takes a function as its first argument.

Why Are They Useful?

Higher-order functions allow for more reusable code, for example the `twice` function shows how we can apply the same function multiple times. The `map` function is another good example of code reusability.

Processing Lists

There is also a video on this section.

The Map Function

The higher-order library function called `map` applies a function to every element of a list.

```
map :: (a -> b) -> [a] -> [b]
```

For example:

```
> map (+1) [1,3,5,7]
[2,4,6,8]
```

```
> map (^3) [1,3,5,7]
[1,27,125,343]

> map reverse ["conversation", "talking", "discussion"]
["noitasrevnoc","gniklat","noissucsid"]
```

The `map` function can be defined in a particularly simple manner using a list comprehension:

```
map :: (a -> b) -> [a] -> [b]
map f xs = [f x | x <- xs]
```

Alternatively, for the purposes of proofs, the map function can also be defined using recursion:

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

The Filter Function

The higher-order library function `filter` selects every element from a list that satisfies a predicate.

```
filter :: (a -> Bool) -> [a] -> [a]
```

For example:

```
> filter even [1..10]
[2,4,6,8,10]
```

Filter can be defined using a list comprehension:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [x | x <- xs, p x]
```

Alternatively, it can be defined using recursion:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs)
   | p x       = x : filter p xs
   | otherwise = filter p xs
```

The Foldr Function

There is also a video on this section.

A number of functions on lists can be defined using the following simple pattern of recursion:

```
f :: Num a => [a] -> a
f []     = v
f (x:xs) = x # f xs
```

The function `f` maps the empty list to some value `v`, and any non-empty list to some function `#` applied to its head and `f` of its tail.

For example:

```
sum :: Num a => [a] -> a
sum []     = 0
sum (x:xs) = x + sum xs

product :: Num a => [a] -> a
product []     = 1
product (x:xs) = x * product xs

and :: [Bool] -> Bool
and []     = True
and (x:xs) = x && and xs
```

The higher-order library function `foldr` (*fold right*) encapsulates this simple pattern of recursion, with the function `#` and the value `v` as arguments.

For example:

```
sum = foldr (+) 0

product = foldr (*) 1

or = foldr (||) False

and = foldr (&&) True
```

`foldr` itself can be defined using recursion:

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

However, it is best to think of `foldr` *non-recursively*, as simultaneously replacing each (`:`) in a list by a given function, and `[]` by a given value, as summarized by:

```
foldr (#) v [x0,x1,...,xn] = x0 # (x1 # (... (xn # v) ...))
```

For example:

```
sum [1,2,3]
= foldr (+) 0 [1,2,3]
= foldr (+) 0 (1:(2:(3:[])))
= 1+(2+(3+0))
= 6
```

Replace each (:) by (+) and [] by 0.

```
product [1,2,3]
= foldr (*) 1 [1,2,3]
= foldr (*) 1 (1:(2:(3:[])))
= 1*(2*(3*1))
= 6
```

Replace each (:) by (*) and [] by 1.

Other Foldr Examples

Even though `foldr` encapsulates a simple pattern of recursion, it can be used to define many more functions than might first be expected.

Recall the length function:

```
length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs
```

For example:

```
length [1,2,3]
= length (1:(2:(3:[])))
= 1+(1+(1+0))
= 3
```

We can replace each (:) by `\_ n -> 1+n` and [] by `0` to get:

```
length :: [a] -> Int
length = foldr (\_ n -> 1+n) 0
```

Now recall the reverse function:

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

For example:

```
reverse [1,2,3]
= reverse (1:(2:(3:[])))
= (([] ++ [3]) ++ [2]) ++ [1]
= [3,2,1]
```

Replace each (:) by `\x xs -> xs ++ [x]` and [] by `[]`. Hence, we have:

```
reverse :: [a] -> [a]
reverse = foldr (\x xs -> xs ++ [x]) []
```

Finally, we note that the append function (++) has a particularly compact definition using `foldr`:

```
(++ ys) = foldr (:) ys
```

Here we replace each (:) by `(:)` and [] by `ys`.

An even more concise definition is the following:

```
(++) = foldr (:)
```

The Foldl Function

There is also a video on this section.

It is also possible to define recursive functions on lists using an operator that is assumed to *associate to the left*.

For example, the function `sum` can be redefined in this manner by using an auxiliary function `sum'` that takes an extra argument `v` that is used to accumulate the final result:

```
sum :: Num a => [a] -> a
sum = sum' 0
      where
         sum' v []     = v
         sum' v (x:xs) = sum' (v+x) xs
```

For example

```
sum [1,2,3]
= sum' 0 [1,2,3]
= sum' (0+1) [2,3]
= sum' ((0+1)+2) [3]
= sum' (((0+1)+2)+3) []
= (((0+1)+2)+3)
= 6
```

Generalizing from the above `sum` example, many functions on lists can be defined using the following simple pattern of recursion.

```
f :: Num a => a -> [a] -> a
f v []     = v
f v (x:xs) = f (v # x) xs
```

That is, the function maps the empty list to the *accumulator* value `v`, and any non-empty list to the result of recursively processing the tail using a new accumulator value obtained by applying an operator `#` to the current value and the head of the list.

The above `sum` function can be re-written using the higher-order library function `foldl` (*fold left*) as:

```
sum :: Num a => [a] -> a
sum = foldl (+) 0
```

Similarly, we can define:

```
product :: Num a => [a] -> a
product = foldl (*) 1

or :: [Bool] -> Bool
or = foldl (||) False

and :: [Bool] -> Bool
and = foldl (&&) True

length :: [a] -> Int
length = foldl (\n _ -> n+1) 0
```

The `foldl` function can be defined using recursion:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f v []     = v
foldl f v (x:xs) = foldl f (f v x) xs
```

However, it is best to think of `foldl` *non-recursively*, in terms of an operator `#` that is assumed to associate to the left, as summarized by the following equation:

```
foldl (#) v [x0,x1,...,xn] = (... ((v # x0) # x1) ...) #xn
```

The Composition Operator

There is also a video on this section.

The library function (.) returns the composition of two functions as a single function.

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \x -> f (g x)
```

That is, `f . g` is read as `f` *composed with* `g`, is the function that takes an argument `x`, applies the function `g` to this argument, and applies the function `f` to the result.

Composition can be used to simplify nested function applications, by reducing the parenthesis and avoid the need to explicitly refer to the initial argument.

For example:

```
odd :: Int -> Bool
odd n = not (even n)
```

Can be defined as:

```
odd :: Int -> Bool
odd = not . even
```

Similarly, the following definitions

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)
```

```
sumsqreven :: Integral a => [a] -> a
sumsqreven ns = sum (map (^2) (filter even ns))
```

can be rewritten more simply as (adding prime to give these different names):

```
twice' :: (a -> a) -> a -> a
twice' f = f . f

sumsqreven' :: Integral a => [a] -> a
sumsqreven' = sum . map (^2) . filter even
```

Other Library Functions

The library function `all` decides if every element of a list satisfies a given predicate.

```
all :: (a -> Bool) -> [a] -> Bool
all p xs = and [p x | x <- xs]
```

For example:

```
> all even [2,4,6,8,10]
True

> all odd [1,3,7,9,10]
False
```

Dually, the library function `any` decides if at least one element of a list satisfies a predicate.

```
any :: (a -> Bool) -> [a] -> Bool
any p xs = or [p x | x <- xs]
```

For example:

```
> any (== ' ') "abc def"
True

> any (> 10) [1,5,4,8,7]
False
```

The library function `takeWhile` selects elements from a list while a predicate holds of all the elements.

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

For example:

```
> takeWhile (/= ' ') "abc def"
"abc"

> takeWhile even [2,4,6,7,8]
[2,4,6]
```

Dually, the function `dropWhile` removes elements while a predicate holds of all the elements.

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

For example:

```
> dropWhile (== 'a') "aaabcadef"
"bcadef"

> dropWhile odd [1,3,5,6,7]
[6,7]
```

Programming Example - Binary String Transmitter

There is also a video on this section.

A binary number is a sequence of zeros and ones, called *bits*, in which successive bits as we move to the left increase in weight by a factor of two.

For example the binary number `1101` can be understood as follows:

```
1101 = (8 * 1) + (4 * 1) + (2 * 0) + (1 * 1)
```

To simplify the definition of certain functions, we assume for the remainder of this example that binary numbers are written in *reverse* order.

For example, `1101` would now be written as `1011`, with successive bits as we move to the right increasing in weight by a factor of two:

```
1011 = (1 * 1) + (2 * 0) + (4 * 1) + (8 * 1)
```

Base Conversion

We begin by importing the library of useful functions on characters and declaring the type of bits as a synonym for the type of integers:

```
import Data.Char
```

```
type Bit = Int
```

We can convert a binary number, represented as a list of bits, to an integer using the `bin2int` function:

```
bin2int :: [Bit] -> Int
bin2int bits = sum [w*b | (w,b) <- zip weights bits]
               where weights = iterate (*2) 1
```

The higher-order library function `iterate` produces an infinite list by applying a function an increasing number of times to a value:

```
iterate f x = [x, f x, f (f x), f (f (f x)), ...]
```

Hence, the expression `iterate (*2) 1` in the above definition of `bin2int` produces the list of weights `[1,2,4,8,...]`, which is then used to compute the weighted sum by means of a list comprehension.

```
> bin2int [1,0,1,1]
13
```

There is, however, a simpler way to define `bin2int` function, based on the algebraic properties of binary numbers. Consider an arbitrary four-bit number [a,b,c,d]. Applying `bin2int` to it produces the following weighted sum:

```
(1 * a) + (2 * b) + (4 * c) + (8 * d)
```

which can be restructured as follows:

```
(1 * a) + (2 * b) + (4 * c) + (8 * d)
= a + (2 * b) + (4 * c) + (8 * d)
= a + 2 * (b + (2 * c) + (4 * d))
= a + 2 * (b + 2 * (c + (2 * d)))
= a + 2 * (b + 2 * (c + 2 * (d + 2 * 0)))
```

From the above result, we see that the conversion can be written as replacing each `cons` by the function that adds its first argument to twice the second argument, and replacing the empty list by zero. Therefore, `bin2int` can be rewritten as (using a slightly different name):

```
bin2int' :: [Bit] -> Int
bin2int' = foldr (\x y -> x + 2*y) 0
```

The opposite function for converting a non-negative integer into binary number can be written as:

```
int2bin :: Int -> [Bit]
int2bin 0 = []
int2bin n = n `mod` 2 : int2bin (n `div` 2)
```

For example:

```
> int2bin 13
[1,0,1,1]
```

We can now define a function `make8` that ensures we have binary numbers of the same length i.e. 8 bits. It either truncates or extends a binary number as appropriate to make it 8 bits long:

```
make8 :: [Bit] -> [Bit]
make8 bits = take 8 (bits ++ repeat 0)
```

For example:

```
> make8 [1,0,1,1]
[1,0,1,1,0,0,0,0]
```

Transmission

There is also a video on this section.

We can now define a function that encodes a string of characters as a list of bits by converting each character into a Unicode number, converting each such number into an eight-bit binary number, and concatenating each of these numbers together to produce a list of bits.

```
encode :: String -> [Bit]
encode = concat . map (make8 . int2bin . ord)
```

For example:

```
> encode "abc"
[1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0,1,1,0,0,0,1,1,0]
```

To decode a list of bits, we firstly define a function `chop8` that chops such a list up into eight-bit binary numbers:

```
chop8 :: [Bit] -> [[Bit]]
chop8 [] = []
chop8 bits = take 8 bits : chop8 (drop 8 bits)
```

Now we can define the `decode` function that chops a list of bits, converts each resulting binary number into a Unicode number and then a character:

```
decode :: [Bit] -> String
decode = map (chr . bin2int) . chop8
```

For example:

```
> decode [1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0,1,1,0,0,0,1,1,0]
"abc"
```

Finally, we can define the function `transmit` that simulates the transmission of a string of characters as a list of bits, using a perfect communication channel that we model using the identity function:

```
transmit :: String -> String
transmit = decode . channel . encode

channel :: [Bit] -> [Bit]
channel = id
```

For example:

```
> transmit "higher-order functions are easy"
"higher-order functions are easy"
```

The above example is actually encapsulating three functions i.e. encoding, transmission and decoding. We can separate the encoding and decoding steps to see what happens in between. The channels is an identity function i.e. it outputs whatever is given to it as input.

```
> encode "higher-order functions are easy"
[0,0,0,1,0,1,1,0,1,0,0,1,0,1,1,0,1,1,1,0,0,1,1,0,0,0,0,1,0,1,1,0,1,0,1,0,0,1,1,0,0,1,0,0,1,1,1,0,1,0,1,1,0,1,0,0,1,1,1

> decode [0,0,0,1,0,1,1,0,1,0,0,1,0,1,1,0,1,1,1,0,0,1,1,0,0,0,0,1,0,1,1,0,1,0,1,0,0,1,1,0,0,1,0,0,1,1,1,0,1,0,1,1,0,1,
"higher-order functions are easy"
```

Exercises

(1) What are higher-order functions that return functions as results better known as?

(2) Express the comprehension `[f x | x <- xs, p x]` using the functions `map` and `filter`. The function type is given as:

```
fun :: Num a => (a -> a) -> (a -> Bool) -> [a] -> [a]
```

For example:

```
> fun (^2) even [1..20]
[4,16,36,64,100,144,196,256,324,400]

> fun (^2) odd [1..20]
[1,9,25,49,81,121,169,225,289,361]
```

(3) Redefine `map f` and `filter p` using `foldr`. For your reference, here are the definitions of `map` and `filter` from lecture notes.

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs

filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs)
   | p x       = x : filter p xs
   | otherwise = filter p xs
```

(4) Define a function `altMap :: (a -> b) -> (a -> b) -> [a] -> [b]` that alternatively applies the two argument functions to successive elements in a list.

For example:

```
> altMap (+10) (+100) [0,1,2,3,4]
[10,101,12,103,14]
```