Haskell

Higher Order Functions

Introduction

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)
```

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

Why Are They Useful?

- ⌘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.

The Map Function

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

```
map :: (a → b) → [a] → [b]
```

Examples

```
> map even [1..4]
[False,True,False,True]
```

```
sum1 x = x+1
```

```
> map sum1 [1,3,5,7]

[2,4,6,8]
```

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

```
map f xs = [f x | x ← xs]
```

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

```
map f []     = []
map f (x:xs) = f x : map f xs
```

Esercizio

- Si consideri una matrice di interi data da una lista di liste
- Si scriva la funzione **`matQuad`** che eleva al quadrato ogni elemento della matrice:

```
> matQuad [[1,2,3],
           [4,3,6],
           [8,1,1]]
[[ 1,4, 9],
 [16,9,36],
 [64,1, 1]]
```

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 p xs = [x | x ← xs, p x]
```

Alternatively, it can be defined using recursion:

```
filter p []     = []
filter p (x:xs)
   | p x        = x : filter p xs
   | otherwise  = filter p xs
```

The Foldr Function

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

```
f []     = v
f (x:xs) = x ⊕ f xs
```

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 []     = 0
sum (x:xs) = x + sum xs
```

v = 0
⊕ = +

```
product []     = 1
product (x:xs) = x * product xs
```

v = 1
⊕ = *

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

v = True
⊕ = &&

The higher-order library function <u>**foldr**</u> (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** <u>non-recursively</u>, as simultaneously replacing each **(:)** in a list by a given function, and **[]** by a given value.

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.

For example:

```
  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
```

Replace each (:) by λ_ n → 1+n and [] by 0.

Hence, we have:

```
length = foldr (λ_ n → 1+n) 0
```

Now recall the reverse function:

```
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 λx xs → xs ++ [x] and [] by [].

Hence, we have:

```
reverse =
   foldr (λx xs → xs ++ [x]) []
```

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

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

Replace each (:) by (:) and [] by ys.

Why Is Foldr Useful?

- Some recursive functions on lists, such as sum, are <u>simpler</u> to define using foldr.
- Properties of functions defined using foldr can be proved using algebraic properties of foldr, such as <u>fusion</u> and the <u>banana split</u> rule.
- Advanced program <u>optimisations</u> can be simpler if foldr is used in place of explicit recursion.

Left Fold

- There is also a left fold:

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

The difference is that in **`foldl`** the value is accumulated “on the left”.

$$\mathtt{foldr}\ \oplus\ a\ [x_1,\ldots,x_n] = x_1\oplus(x_2\oplus(\ \ldots\ (x_n\oplus a)))$$

$$\mathtt{foldl}\ \oplus\ a\ [x_1,\ldots,x_n] = (((a\oplus x_1)\oplus x_2)\ \ldots\ \oplus x_n)$$

Exercise

- Count the number of vowels in a string using folds
- Useful function: **`elem x xs`** checks if **`x`** is in list **`xs`**

```
> vocali "Haskell"
2
```

Function composition

The library function **`(.)`** returns the <u>composition</u> of two functions as a single function.

```
(.)   :: (b → c) → (a → b) → (a → c)
f . g  = λx → f (g x)
```

For example:

```
odd :: Int → Bool
odd  = not . even
```

Function composition

- It composes functions in a readable manner

`f(g(h(k(x))))`

vs

`(f.g.h.k)(x)`

- Note that usually functions associate to the left

```
>not even 2
ERROR

>(not.even) 2
False
```

Example

- Given a list of numbers, create a list of all negated absolute values using **`map`**

```
> f [1,-2,-5,0,3]
[-1,-2,-5,0,-3]
```

- Useful functions: **`abs`** and **`negate`**

The library function <u>all</u> 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
```

Dually, the library function <u>any</u> 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
```

The library function <u>takeWhile</u> 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"
```

Dually, the function <u>dropWhile</u> 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 (== ' ') "   abc"

"abc"
```

Exercises

- Express the comprehension

  `[f x | x ← xs, p x]`

  using the functions **`map`** and **`filter`**.

- Write a function zipWith’

```
zipWith’ :: (a -> b -> c) -> [a] -> [b] -> [c]
```

with the following behavior

```
>zipWith' (+) [4,2,5,6] [2,6,2,3]
[6,8,7,9]
```

Esercizio (estratto dal compito 29 giu 2016)

- si scriva una funzione di ordine superiore

```
maxf :: Ord a => (t -> a) -> [t] -> t
```

- che prende come parametri una funzione *f* e una lista *xs* e fornisce l'elemento *x* della lista *xs* che massimizza la funzione *f* (ossia il valore *x* per cui *f*(*x*) è massimo).

Create a password strength checker

- Create a password strength checker using higher-order functions
- A strong password has
  - at least 15 characters
  - uppercase letters
  - lowercase letters
  - numbers

```
Prelude> :t strong
strong :: String -> Bool

Prelude> strong "sup3rL33Tpassw0rd"
True
```

- The function **`remdups`** removes adjacent duplicates from a list. For example,

```
>remdups [1, 2, 2, 3, 3, 3, 1, 1]

[1, 2, 3, 1]
```

- Define **`remdups`** using **`foldr`** . Give another definition using **`foldl`** .

Un po' più difficili …

- Redefine `map f` and `filter p` using `foldr`.

These slides were adapted from the material of the book

Graham Hutton, Programming in Haskell, Cambridge University Press, 2nd edition, 2016