PROGRAMANDO EN HASKELL

Capítulo 7 - Funciones de alto orden

Introducción

A una función se le llama de alto orden si toma una función como argumento o regresa una función como resultado.

```
twice :: (a → a) → a → a
twice f x = f (f x)
```

twice es de alto orden porque toma una función como su primer argumento.

¿Por qué son útiles?

- Idiomas comunes de programación pueden ser encodificados como funciones dentro del lenguaje en sí mismo.

- Lenguajes de Dominio Especifico pueden definirse como colecciones de funciones de alto orden.

- Las propiedades algebraicas de las funciones de alto orden pueden utilizarse para el razonamiento sobre los programas.

La Función Map

La función map, de alto orden, y presente en la librería estándar, aplica una función a cada elemento de una lista.

```
map :: (a → b) → [a] → [b]
```

Por ejemplo:

```
> map (+1) [1,3,5,7]

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

La función map puede definirse de una manera particularmente simple usando listas comprendidas:

```
map f xs = [f x | x ← xs]
```

Alternativamente, con fines de prueba, la función map puede definirse usando recursión:

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

La Función Filter

La función filter, de alto orden, y presente en la librería estándar, selecciona cada elemento de una lista que satisfaga un predicado.

```
filter :: (a → Bool) → [a] → [a]
```

Por ejemplo:

```
> filter even [1..10]

[2,4,6,8,10]
```

Filter puede definirse usando listas comprendidas:

```
filter p xs = [x | x ← xs, p x]
```

Alternativamente, se puede definir con recursión:

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

La Función Foldr

Se pueden definir varias funciones sobre listas utilizando el siguiente patrón simple de recursión:

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

f mapea la lista vacía a algún valor v, y cualquier lista no vacía a alguna función ⊕ aplicada su cabezal y 'f' a los elementos restantes de la lista.

Por ejemplo:

```
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
⊕ = &&

La función de alto orden foldr (fold right), en la librería estándar, encapsula este patrón simple de recursión, con la función $\oplus$ y el valor v como argumentos.

Por ejemplo:

```
sum = foldr (+) 0

product = foldr (*) 1

or = foldr (||) False

and = foldr (&&) True
```

Foldr en sí misma puede definirse con recursión:

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

No obstante, es mejor pensar en foldr de manera no recusiva, como si de forma simultánea reemplazaramos cada (:) en una lista con una función dada, y [] por un valor dado.

Por ejemplo:

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

Reemplaza cada (:) por (+) y [] con 0.

Por ejemplo:

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

Reemplaza cada (:) por (*) y [] con 1.

Otros Ejemplos Con Foldr

Aunque foldr encapsula un patrón simple de recursión, puede ser usado para definir muchas mas funciones de las que uno esperaría en un principio.

Recuerda la función length:

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

Por ejemplo:

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

Reemplaza cada (:) por λ_ n → 1+n y [] con 0.

Así, tenemos:

```
length = foldr (λ_ n → 1+n) 0
```

Ahora recuerda la función reverse:

```
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Por ejemplo:

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

Reemplaza cada (:) por $\lambda x\ xs \rightarrow xs$ ++ [x] y [] con [].

Así, tenemos:

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

Finalmente, nos damos cuenta que la función append (++) tiene una definición particularmente compacta usando foldr:

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

Reemplaza cada (:) por (:) y [] con ys.

¿Por qué es útil Foldr?

- Algunas funciones recursivas sobre listas, como sum, son mas simples de definir con foldr.

- Las propiedades de las funciones que usan foldr pueden ser probadas utilizando las propiedades algebraicas de foldr, tales como fusion y la regla banana split.

- Las Optimizaciones avanzadas de los programas pueden ser simples si se utiliza foldr en lugar de recursión explícita.

Otras Funciones en la Librería

La función (.), presente en la librería, regresa la composición de dos funciones como una sola función.

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

Por ejemplo:

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

La función all, también en la librería, decide si cada elemento de una lista satisface un predicado dado.

```
all :: (a → Bool) → [a] → Bool
all p xs = and [p x | x ← xs]
```

Por ejemplo:

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

True
```

La función any (dual de all), presente en la librería, decide si al menos un elemento de una lista satisface un predicado.

```
any :: (a → Bool) → [a] → Bool
any p xs = or [p x | x ← xs]
```

Por ejemplo:

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

True
```

La función de librería takeWhile selecciona elementos de una lista mientras un predicado es verdadero para todos los elementos.

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

Por ejemplo:

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

"abc"
```

De manera dual, la función dropWhile borra elementos mientras un predicado es verdadero para todos los elementos.

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

Por ejemplo:

```
> dropWhile (== ' ') "   abc"

"abc"
```

Ejercicios

(1) ¿Las funciones de alto orden que regresan funciones como resultado son mejor conocidas cómo?

(2) Expresa la lista comprendida [f x | x $\leftarrow$ xs, p x] usando las funciones map y filter.

(3) Redefine map f y filter p usando foldr.

Fuente.

Profesor Graham Hutton.
http://www.cs.nott.ac.uk/~pszgmh/

Traducción al español.

Caleb Josue Ruiz Torres.
https://www.calebjosue.com