Orden superior

Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

Orden Superior

- Una función es de `orden superior` si toma una función como argumento o devuelve una función como resultado.
- `(dosVeces f x)` es el resultado de aplicar `f` a `f x`. Por ejemplo:

```
dosVeces (*3) 2            ==  18
dosVeces reverse [2,5,7]   ==  [2,5,7]

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

- Prop: `dosVeces reverse = id`, donde `id` es la función identidad.

```
id :: a -> a
id x =  x
```

Usos de las funciones de orden superior

- Definición de patrones de programación.
  - Aplicación de una función a todos los elementos de una lista.
  - Filtrado de listas por propiedades.
  - Patrones de recursión sobre listas.
- Diseño de lenguajes de dominio específico:
  - Lenguajes para procesamiento de mensajes.
  - Analizadores sintácticos.
  - Procedimientos de entrada/salida.
- Uso de las propiedades algebraicas de las funciones de orden superior para razonar sobre programas.

La función `map`

- `(map f xs)` es la lista obtenida aplicando `f` a cada elemento de `xs`. Por ejemplo:

```
map (*2) [3,4,7]      == [6,8,14]
map sqrt [1,2,4]      == [1.0,1.4142135623731,2.0]
map even [1..5]       == [False,True,False,True,False]
```

- Definición de map por comprensión:

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

- Definición de map por recursión:

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

Relación entre sum y map

- La función sum:

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

- Propiedad:

```
sum (map (2*) xs) = 2 * sum xs
```

- Comprobación con QuickCheck:

```
prop_sum_map :: [Int] -> Bool
prop_sum_map xs = sum (map (2*) xs) == 2 * sum xs

ghci> quickCheck prop_sum_map
+++ OK, passed 100 tests.
```

La función `filter`

- `filter p xs` es la lista de los elementos de `xs` que cumplen la propiedad `p`. Por ejemplo:

```
filter even [1,3,5,4,2,6,1] == [4,2,6]
filter (>3) [1,3,5,4,2,6,1] == [5,4,6]
```

- Definición de filter por comprensión:

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

- Definición de filter por recursión:

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

Uso conjunto de map y filter

- `sumaCuadradosPares xs` es la suma de los cuadrados de los números pares de la lista `xs`. Por ejemplo:

```
sumaCuadradosPares [1..5]  ==  20

sumaCuadradosPares :: [Int] -> Int
sumaCuadradosPares xs = sum (map (^2) (filter even xs))
```

- Definición por comprensión:

```
sumaCuadradosPares' :: [Int] -> Int
sumaCuadradosPares' xs = sum [x^2 | x <- xs, even x]
```

Predefinidas de orden superior para procesar listas

- `all p xs` se verifica si todos los elementos de `xs` cumplen la propiedad `p`. Por ejemplo:

```
all odd [1,3,5]  ==  True
all odd [1,3,6]  ==  False
```

- `any p xs` se verifica si algún elemento de `xs` cumple la propiedad `p`. Por ejemplo:

```
any odd [1,3,5]  ==  True
any odd [2,4,6]  ==  False
```

- `takeWhile p xs` es la lista de los elementos iniciales de `xs` que verifican el predicado `p`. Por ejemplo:

```
takeWhile even [2,4,6,7,8,9] == [2,4,6]
```

- `dropWhile p xs` es la lista `xs` sin los elementos iniciales que verifican el predicado `p`. Por ejemplo:

```
dropWhile even [2,4,6,7,8,9] == [7,8,9]
```

Funciones lambda (anónimas)

La expresión

```
\x1 x2 ...xn -> cuerpo
```

representa a una función anónima (también denominadas `lambda`); donde x1 x2 ...xn son los argumentos de dicha función.

- Son funciones que se construyen sin nombrarlas.
- Solo existen en el contexto donde se definen.
- El cuerpo se define con solo un patrón.
- Funciones de un solo uso.

Por ejemplo

```
*Main> (\x -> x*x + 1) 3
10
```

Funciones lambda (anónimas)

Ejemplo de función sin funciones lambda:

```
elevaCuadrado :: [Int] -> [Int]
elevaCuadrado xs = map f xs
  where f x = x*x
```

Ejemplo de función usando funciones lambda:

```
elevaCuadrado' :: [Int] -> [Int]
elevaCuadrado' xs = map (\x -> x*x) xs
```

Funciones lambda (anónimas)

Otros ejemplos:

```
*Main> map (\x -> 2*x + 1) [1..5]
[3,5,7,9,11]
*Main> filter (\(_,y) -> y > 2) [(4,5),(6,(-1)),(0, 8)]
[(4,5),(0,8)]
*Main> all (\xs -> (not (null xs))) [[4,5,6],[],[1..10]]
False
```

Plegado por la derecha: foldr

Esquema básico de recursión sobre listas

- Ejemplos de definiciones recursivas:

```
sum []          = 0
sum (x:xs)      = x + sum xs
product []      = 1
product (x:xs) = x * product xs
or []           = False
or (x:xs)       = x || or xs
and []          = True
and (x:xs)      = x && and xs
```

- Esquema básico de recursión sobre listas:

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

El patrón foldr

- Redefiniciones con el patrón `foldr`

```
sum     = foldr (+) 0
product = foldr (*) 1
or      = foldr (||) False
and     = foldr (&&) True
```

- Definición del patrón `foldr`

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

Visión no recursiva de foldr

- Cálculo con (+):

```
sum [2,3,5]
= foldr (+) 0 [2,3,5]          [def. de sum]
= foldr (+) 0 2:(3:(5:[]))     [notación de lista]
=             2+(3+(5+0))      [sustituir (:) por (+) y []  por 0]
= 10‘                          [aritmética]
```

- Cálculo con (*):

```
product [2,3,5]
= foldr (*) 1 [2,3,5]          [def. de sum]
= foldr (*) 1 2:(3:(5:[]))     [notación de lista]
=             2*(3*(5*1))      [sustituir (:) por (*) y []  por 1]
= 30                           [aritmética]
```

- Cálculo de foldr f v xs:
  - Sustituir en `xs` los `(:)` por `f` y `[]` por `v`.

Definición de la longitud mediante foldr

- Ejemplo de cálculo de la longitud:

```
longitud [2,3,5]
= longitud 2:(3:(5:[]))
=          1+(1+(1+0))      [Sustituciones]
= 3
```

- Sustituciones:
  - los `(:)` por `(\x y -> 1+y)`
  - la `[]` por 0
- Definición de `length` usando `foldr`

```
longitud :: [a] -> Int
longitud = foldr (\x y -> 1+y) 0
```

Definición de la inversa mediante foldr

- Ejemplo de cálculo de la inversa:

```
inversa [2,3,5]
= inversa 2:(3:(5:[]))
=         (([] ++ [5]) ++ [3]) ++ [2]   [Sustituciones]
= [5,3,2]
```

- Sustituciones:
  - los `(:)` por `(\x y -> y ++ [x])`
  - la `[]` por `[]`
- Definición de inversa usando foldr

```
inversa :: [a] -> [a]
inversa = foldr (\x y -> y ++ [x]) []
```

Definición de la concatenación mediante foldr

- Ejemplo de cálculo de la concatenación:

```
conc [2,3,5] [7,9]
= conc 2:(3:(5:[])) [7,9]
=      2:(3:(5:[7,9]))       [Sustituciones]
= [2,3,5,7,9]
```

- Sustituciones:
  - los `(:)` por `(:)`
  - la `[]` por `ys`
- Definición de la concatenación usando `foldr`

```
conc xs ys = (foldr (:) ys) xs
```

Plegado por la izquierda: foldl

Definición de suma de lista con acumuladores

- Definición de suma con acumuladores:

```
suma :: [Integer] -> Integer
suma = sumaAux 0
   where sumaAux v []     = v
sumaAux v (x:xs) = sumaAux (v+x) xs
```

- Cálculo con suma:

```
suma [2,3,7]
= sumaAux 0 [2,3,7]
= sumaAux (0+2) [3,7]
= sumaAux 2 [3,7]
= sumaAux (2+3) [7]
= sumaAux 5 [7]
= sumaAux (5+7) []
= sumaAux 12 []
= 12
```

Plegado por la izquierda: foldl

Patrón de definición de recursión con acumulador

- Patrón de definición (generalización de `sumaAux`):

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

El patrón foldl

- Redefiniciones con el patrón `foldl`

```
suma    = foldl (+) 0
product = foldl (*) 1
or      = foldl (||) False
and     = foldl (&&) True
```

- Definición del patrón `foldl`

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

Diferencia entre foldr y foldl

- Diferencia entre `foldr` y `foldl`:

```
(foldr (-) 0) [3,4,2] =     3-(4-(2-0)) = 1
(foldl (-) 0) [3,4,2] = ((0-3)-4)-2     = -9
```

Variantes: foldr1 y foldl1

- `foldr1` y `foldl1`: versiones de `foldr` y `foldl` donde se trabaja con listas no vacías, no se da un valor inicial sino que se empieza por el último o primer elemento (respectivamente).

```
maximum' = foldr1 (\x acc -> if x > acc then x else acc)

sum' = foldl1 (+)
```

Variantes: scanr y scanl

- `scanr`, `scanl`, son como `foldr` y `foldl` (respectivamente), solo que devuelven todos los estados intermedios acumulados en forma de una lista. También existen las versiones `scanr1`, `scanl1`.

```
> scanl (+) 0 [3,5,2,1]
[0,3,8,10,11]
> scanr (+) 0 [3,5,2,1]
[11,8,3,1,0]
> scanl1 (\acc x -> if x > acc then x else acc)
   [3,4,5,3,7,9,2,1]
[3,4,5,5,7,9,9,9]
> scanl (flip (:)) [] [3,2,1]
[[],[3],[2,3],[1,2,3]]
```

Composición de funciones

- Definición

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

- Uso de composición para simplificar definiciones
  - Definiciones sin composición:

```
par n                    = not (impar n)
doVeces f x              = f (f x )
sumaCuadradosPares ns = sum (map (^2) (filter even ns))
```

  - Definiciones con composición:

```
par                  = not . impar
dosVeces f           = f . f
sumaCuadradosPares = sum . map (^2) . filter even
```

Composición de una lista de funciones

- La función identidad:

```
id :: a -> a
id =  \x -> x
```

- (composicionLista fs) es la composición de la lista de funciones fs. Por ejemplo:

```
composicionLista [(*2),(^2)] 3       ==  18
composicionLista [(^2),(*2)] 3       ==  36
composicionLista [(/9),(^2),(*2)] 3  ==  4.0
```

```
composicionLista :: [a -> a] -> (a -> a)
composicionLista =  foldr (.) id
```

- Ejercicio propuesto: adaptarlo a plegado por la izquierda.
- Ejemplo extenso: Codificación binaria y transmisión de cadenas

Funciones parciales (currying)

- Currying viene del matemático Haskell Curry.
- La función `curry` y `uncurry` convierten una función cuyos parámetros vienen dados en una tupla a una función definida parcialmente con $(->)$ y viceversa:

```
f :: a -> (b -> c)
g :: (a, b) -> c
f = curry g
g = uncurry f
```

- Todas las funciones que aceptan varios parámetros como hemos visto hasta ahora están curryficadas. Por ejemplo, la función "f" del ejemplo anterior.
- Ejemplos:

```
> let f (x,y) = x+y
> f (2,3)
5
> (curry f) 2 3
5
```

Funciones parciales

- Si llamamos a una función con pocos parámetros, obtenemos una función parcialmente aplicada que toma tantos parámetros como los que hemos dejado fuera.

```
> max 4 5
5
> (max 4) 5
5
> :t max
max :: (Ord a) => a -> a -> a
max :: (Ord a) => a -> (a -> a)
> :t max 4
max :: (Ord a, Num a) => a -> a
```

Funciones infijas parciales

- Lo mismo ocurre con las funciones infijas, pueden ser parcialmente aplicadas usando secciones.
- Para seccionar una función infija, simplemente se rodea con paréntesis y se provee un parámetro en uno de los lados.
- Esto crea una función que toma un parámetro y lo aplica en el lado donde falta un operando.

```
dividePorDiez :: (Floating a) => a -> a
dividePorDiez = (/10)
```

Inversión de parámetros

- Puede ser útil la función `flip`:

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f = g
  where g x y = f y x
```

- Por ejemplo:

```
> flip zip [1,2,3,4,5] "hello"
[('h',1),('e',2),('l',3),('l',4),('o',5)]
> zipWith (flip div) [2,2..] [10,8,6,4,2]
[5,4,3,2,1]
```

Funciones lambda y parcialización

Se pueden usar expresiones lambda para resaltar la parcialización.

Ejemplo *suma*:

- Sin lambda:

```
suma x y = x + y
```

- Con lambda:

```
suma = \x -> (\y -> y + x)
```

Ejemplo *identidad*:

- Sin lambda:

```
id x = x
```

- Con lambda:

```
id x = \_ -> x
```

Funciones lambda y parcialización

Ejemplos con operadores:

```
(*) = \x -> (\y -> x * y)
(x*) = \y -> x * y
(*y) = \x -> x * y
```

Más información sobre este tema en esta presentación (prestar especial atención al apartado sobre **secciones** y a lo referente a funciones lambda).

Clausura y variables libres

Una **clausura (closure)** es una función que hace uso de **variables libres** en su definición, lo cual implica cerrar cierto entorno alrededor de la definición de la función.

```
> f x = (\y -> x + y)
```

`f` devuelve una clausura, ya que la variable `x` es una variable libre: se usa en la definición de la función lambda (devuelta por f) pero viene dada desde fuera de dicha definición.
Ejemplos de clausuras en otros lenguajes en wikipedia y aquí.

Bibliografía

J.A. Alonso. *Funciones de orden superior*. Tema 7, Informática, Grado de Matemáticas 2019.
http://www.cs.us.es/ jalonso/cursos/i1m-18/temas/tema-7.html

R. Bird. *Introducción a la programación funcional con Haskell*. Prentice Hall, 2000.
Capítulo 4: Listas

G. Hutton *Programming in Haskell*. Cambridge University Press, 2007.
Chapter 7: Higher-order functions

B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. Razonando con Haskell. Thompson, 2004..
Capítulo 8: Funciones de orden superior y polimorfismo

S. Thompson. Haskell: The Craft of Functional Programming, Second Edition. Addison-Wesley, 1999.
Chapter 9: Generalization: patterns of computation
Chapter 10: Functions as values