Clase Práctica 1 : Programación Funcional

Tomás Felipe Melli

July 9, 2025

Índice

1 Currificación y aplicación parcial

Miremos estas funciones :

$$prod :: (Int, Int) \to Int$$
$$prod\ (x, y) = x * y$$

$$prod' :: Int \to (Int \to Int)$$
$$(prod'\ x)\ y = x * y$$

Los paréntesis en violeta pueden o no ir ya que Haskell asocia a derecha. Con respecto a los paréntesis verdes, la aplicación asocia a izquiera, por tanto, tampoco estaría mal sacarlos.

Las funciones siempre toman un único argumento ! Qué hacen estas funciones entonces ?

- Por un lado `prod` recibe una tupla de dos elementos.
- Por el otro, `prod'` es una función que toma un x de tipo Int y devuelve una función de tipo $Int -> Int$, cuyo comportamiento es tomar un entero y multiplicarlo por x. En particular, (prod' 2) es la función que duplica. Una definición equivalente de `prod'` usando **funciones anónimas**:

  $$prod'\ x = \backslash y \to x * y$$

  Esta es la llamada **notación lambda**. Es útil cuando queremos pasar funciones como argumentos.

Ahora bien, decimos que la *versión currificada de prod es prod'*

1.1 curry - uncurry

Nos piden definir dos funciones :

1. `curry` que lo que hace es devolver la versión currificada de una función no currificada

```haskell
curry :: ((a,b) -> c) -> a -> b -> c
curry f x y = f (x,y)
-- curry f es de tipo a -> b -> c
-- x es de tipo a
-- y es de tipo b
-- haskell lee (curry f) x y

-- Alternativas:
-- curry f = \x -> \y -> f (x,y)
-- curry f = \x y -> f (x,y)
```

   Podemos pensar en $curry :: ((a, b) -> c) -> (a -> b -> c)$ para ver más explícitamente.

2. `uncurry` que dada una función currificada devuelve su versión no currificada

```haskell
uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f = \(x,y) -> f x y
-- = uncurry f (x,y) = f x y
```

   Podemos pensar en $uncurry :: (a -> b -> c) -> ((a, b) -> c)$ para ver más explícitamente

1.2 Aplicación parcial

Con la currificación podemos hacer aplicación o evaluación parcial : obtener funciones y pasarlas como parámetro como vemos a continuación :

```haskell
prod :: Int -> Int -> Int
prod x y = x * y

doble x = prod 2 x
```

- Cuál es el tipo de `doble`? `Int` $\to$ `Int`
- ¿Qué pasa si cambiamos la definición por`doble = prod 2`? Anda, e incluso vamos a preferir notarlo de esta manera.

- Qué significa `(+) 1` ? Es la función `Int` → `Int` que suma uno.

Nos piden definir :

- `Triple`

```
triple :: Float -> Float
triple = (*) 3
-- Haskell inifere el tipo Num a => a -> a
-- Esto est  evaluado parcialmente
```

- `esMayorDeEdad`

```
esMayorDeEdad :: Int -> Bool
esMayorDeEdad = (<) 17
-- = (17<)
-- > :: Int -> Int -> Bool ; (>) 5 3 = True
```

2 Funciones útiles

Nos piden implementar y dar los tipos de :

- `(.)` que compone dos funciones.

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

- `flip` que invierte el orden de los argumentos de una función

```
flip :: (a -> b -> c) -> (b -> a -> c)
-- Tambien podr a haber puesto flip ((a->b)->c) -> (b->a->c)
flip f = \x -> \y -> f y x
-- Alternativas:
-- flip f = \x y -> f y x
-- flip f x y = f y x
```

- `($)` que aplica una función a un argumento

```
($) :: (a -> b) -> a -> b
($) f = f
-- Toma una funci n y argumento, y devuelve la funci n aplicada al argumento.
-- Sirve para omitir par ntesis. Ejemplo: f(g(h x)) = f $ g $ h $ x.
-- Como $ asocia a derecha, haskell lee f $ (g $ (h$x)).
```

- `const` que dado un valor, retorna una función constante, que devuelve siempre ese valor

```
const :: a -> (b -> a)
const x = \_ -> x
-- Alternativas:
-- const x _ = x
-- const = \x -> \_ -> x
-- const x = \y ->  x
-- const = \x -> \y -> c
-- OJO: const \x -> x es la identidad
```

Qué hace `flip ($) 0` ?

Esto es `flip ($) 0 f = ($) f 0 = f 0` . O sea, al pasarle una función evalúa esa función en 0.

Qué hace `(==0) . flip(mod 2)`?

Se fija si un número es par. La aplicación de una función tiene mayor precedencia que `(.)`

3 Listas

Tenemos 3 formas de definir listas

1. **Por extensión**, que es dar explícitamente la lista, escribiendo sus elementos. Por ejemplo `[1,2,3,4,5]`.
2. **Secuencias**, dar un progresión aritmética en un rango particular, como sigue `[1..10]` que nos devolverá una lista del 1 al 10([1,2,3,4,5,6,7,8,9,10]).
3. **Por comprensión**, se define como

   ```
   [expresión | selectores, condiciones]
   ```

   Por ejemplo, `[(x,y) | x <- [0..5], y <- [0..3], x + y == 4]`. Esta lista tendrá todos los pares (1,3), (2,2), (3,1) y (4,0). Vale también poner algo como `[(x,y) | x <- [0..5], y <- [0..x]]`

En Haskell podemos trabajar con listas infinitas también.

- `naturales = [1..]`
  1, 2, 3, 4, ...
- `multiplosDe3 = [0,3..]`
  0, 3, 6, 9, ...
- `repeat ‘‘hola’’`
  “hola”, “hola”, “hola”, “hola”, ...
- `primos = [n | n <- [2..], esPrimo n]`
  (asumiendo `esPrimo` definida) 2, 3, 5, 7, ...
- `infinitosUnos = 1 : infinitosUnos`
  1, 1, 1, 1, ...

Cómo es posible trabajar con listas infinitas sin que se cuelgue?

Esto es por la **evaluación lazy**.

4 Lazy evaluation

En Haskell, la evaluación de las funciones sigue el modelo de evaluación Lazy. Por tanto, las expresiones no se evalúan de inmediato, **sólo cuando sus valores son realmente necesarios**. Si una función nunca usa un argumento, nunca se evalúa. Las expresiones se evalúan de manera no estricta y de afuera hacia adentro en la mayoría de los casos pero con una **evaluación diferida**. Esto significa que Haskell intenta primero aplicar funciones antes de evaluar sus argumentos. Miremos detenidamente este ejemplo :

```
1  take :: Int -> [a] -> [a]
2  take 0 = []
3  take [] = []
4  take n (x:xs) = x : take (n-1) xs
5
6  infinitosUnos :: [Int]
7  infinitosUnos = 1 : infinitosUnos
8
9  nUnos :: Int -> [Int]
10 nUnos n = take n infinitosUnos
```

Si ejecutamos `nUnos 2` sucede

```
nUnos 2 → take 2 infinitosUnos
        → take 2 (1 :  infinitosUnos)
        → 1 :  take (2 - 1) infinitosUnos
        → 1 :  take 1 infinitosUnos
        → 1 :  take 1 (1 :  infinitosUnos)
        → 1 :  1 :  take (1 - 1) infinitosUnos
        → 1 :  1 :  take 0 infinitosUnos
        → 1 :  1 :  []
```

Evalúa lo que puede y está más a la izquierda posible. No se cuelga por la evaluación lazy. Haskell en este escenario resuelve la subexpresión más interna.

Qué pasaría si optásemos por otra estrategia de reducción?

Si para algún término existe una reducción finita, entonces la estrategia de reducción lazy termina

5 Funciones de alto orden

5.1 mejorSegun

Nos piden definir:

- máximo

```
maximo :: Ord a => [a] -> a
maximo [x] = x
maximo (x:xs) = if x > maximo xs then x else maximo xs
```

- mínimo

```
minimo :: Ord a => [a] -> a
-- minimo [x] = x
-- minimo (x:xs) = if x < minimo xs then x else minimo xs
```

- listaMásCorta

```
listaMasCorta :: [[a]] -> [a]
listaMasCorta [xs] = xs
listaMasCorta (xs:xss) = if length xs < length (listaMasCorta xss) then xs else listaMasCorta xss

-- Obs: toma bien el length xs < length (listaMasCorta xss) y no lo toma como (length xs < length) (
    listaMasCorta xss)
-- porque la aplicaci n de funciones tiene precedencia m s alta que los operadores como <, -, *, /,
     ^, ==, >, &&, ||, :, ++, =, <- etc.
```

Como la idea es siempre la misma, nos preguntamos si se puede generalizar...

```
mejorSegun :: (a -> a -> Bool) -> [a] -> a
-- Entra un predicado que devuelve true si el primer par metro es mejor que el segundo
mejorSegun _ [x] = x
mejorSegun p (x:xs) = if p x rec then x else rec
    where rec = mejorSegun p xs
-- mejorSegun p (x:xs) = if (p x (mejorSegun p xs)) then x else (mejorSegun p xs)

maximo :: Ord a => [a] -> a
maximo = mejorSegun (>)

minimo :: Ord a => [a] -> a
maximo = mejorSegun (<)

listaMasCorta :: [[a]] -> [a]
listaMasCorta = mejorSegun (\x1 x2 -> length x1 < length x2)
```

5.2 filter

```
filter :: (a -> Bool) -> [a] -> [a]
filter [] = []
filter p (x:xs) = if p x then x : filter p xs else filter p xs
```

Con esto en mente, nos piden definir usando filter :

- deLongitudN

```
deLongitudN :: Int -> [[a]] -> [[a]]
deLongitudN n = filter (\x -> length x == n)
-- deLongitudN n xs = filter (\x -> length x == n) xs
```

- soloPuntosFijosEn que dados un n y una lista de funciones, deja las funciones que al aplicarlas a n dan n

```
soloPuntosFijosEnN :: Int -> [Int->Int] -> [Int->Int]
soloPuntosFijosEnN n = filter (\f -> f n == n)
```

5.3 Transformar elementos de una lista `map`

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

Usando `map` nos piden

- `reverseAnidado` que dada una lista de strings devuelve una lista con cada string dado vuelta y la lista completa dada vuelta.

```haskell
-- Podemos usar reverse :: [a] -> [a]
reverseAnidado :: [[Char]] -> [[Char]]
reverseAnidado = reverse . (map reverse)
-- Es lo mismo que:
-- reverseAnidado xs = reverse (map reverse xs). Ac  no podes sacar ni el xs ni los par ntesis
--                   = \XS -> reverse (map reverse xs)
--                   = reverse . (map reverse) -- Dado XS aplico map reverse y despu s reverse
```

- `paresCuadrador` que dada una lista de enteros devuelve una lista con los cuadrados de los números pares y los impares sin modificar

```haskell
paresCuadrados :: [Int] -> [Int]
paresCuadrados = map (\x -> if even x then x*x else x)
```

5.4 Similitudes entre `map` y `filter`

```haskell
filter ::  (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs) = if p x then x :  filter p xs
                         else filter p xs

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

→ En el caso base devolvemos un valor determinado.

En el caso recursivo devolvemos algo en función de:

- → La cabeza de la lista.
- → El llamado recursivo sobre la cola de la lista.

Podemos generalizar ?

5.5 `foldr`

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

→ z es el valor que devolvemos para una lista vacía.

- f es una función que computa el resultado sobre la lista entera a partir de:
  - → La cabeza de la lista.
  - → El llamado recursivo sobre la cola de la lista.

Si `xs = [x1, x2, x3]` entonces `foldr f z xs = f x1 (f x2 (f x3 z))`. Lo cual es equivalente con notación infija a `foldr *z xs = x1 *(x2 *(x3 *z))`
`foldr` toma una función que combina la cabeza de la lista con el resultado de la recursión sobre la cola de la lista, luego, el caso base luego la lista sobre la cuál se hace la recursión.

```
foldr _ z [] = z -- valor para el caso base
foldr f z (x:xs) = f x (foldr f z xs) -- da el resultado para toda la lista usando f para combinar la
    cabeza de la lista con el resultado de la recursi n sobre la cola de la lista
```

Reescribimos `filter` y `map` con `foldr`

```
filter :: (a -> Bool) -> [a] -> [a]
filter p xs =
foldr (\x r -> if p x then x:r else r) [] xs

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

Es necesario el argumento `xs` ?

La respuesta es **no**. Por tanto,

```
filter :: (a -> Bool) -> [a] -> [a]
filter p =
foldr (\x r -> if p x then x:r else r) []

map :: (a -> b) -> [a] -> [b]
map f  = foldr (\x r -> f x : r) []
```

5.6 `listaComp`

```
-- listaComp f xs p = [f x | x <- xs, p x]
-- Ahora usando map y filter
-- Agarra [a], la filtra segun (a->Bool), y a lo que queda le aplica (a->b) a cada elemeneto.
listaComp :: (a->b) -> [a] -> (a->Bool) -> [b]
listaComp f xs p = map f (filter p xs)
```