Clase Práctica 2 : Programación Funcional

Tomás Felipe Melli

July 9, 2025

Índice

1 Repaso

Queremos definir `maximoL` que tiene como precondición que la lista sea no vacía. Vamos a usar `max` del preludio.

```
-- pre: lista no vac a
maximoL :: (Ord a , Num a ) = > [a] -> a
maximoL xs = foldr max 0 xs
```

Con esta pre, podríamos aplicar el siguiente patrón

```
-- pre: lista no vac a
maximoL :: (Ord a ) = > [a] -> a
maximoL (x:xs) = foldr max x xs
```

Dentro de la familia de *folds* sobre listas existen algunas funciones adicionales, como `foldr1`

```
-- pre: lista no vac a
foldr1 :: (a -> a -> a) -> [a] -> a
```

Por tanto, reescribimos `maximoL` como :

```
maximoL = foldr1 max
```

El tipo de `foldr1` es diferente al de `foldr`. **El caso base de `foldr1` devuelve un elemento de la lista (importante: el tipo debe ser del tipo de la lista).**
Las variantes de `foldr` abstraen el **esquema de recursión estructural**.

Y si no está hecha con `foldr` la función ?

1.1 take’

Miremos

```
take’ :: [a] -> Int -> [a]
take’ []     n = []
take’ (x:xs) n = if n == 0
                 then []
                 else x : take’ xs (n-1)
```

Este esquema **es estructural ya que se usa el argumento inductivo de la lista (la cola)**.
Con aplicación parcial, podemos retornar como caso base otra función, y esto permite usar `foldr`

```
take’ :: [a] -> Int -> [a]
take’ []     = const []
take’ (x:xs) = \n -> if n == 0
                     then []
                     else x : take’ xs (n-1)

take’ :: [Int] -> Int -> [Int]
take’ = foldr (\x rec -> \n -> if n == 0
                               then []
                               else x : rec (n-1)) (const [])
```

Qué pasa si dejamos el tipo original de take ?

```
take ::  Int -> [a] -> [a]
take 0 = \xs -> []
take n = \xs -> if null xs then [] else x : take (n-1) tail xs
-- O sea, take n = foldNat (const [])(\rec -> \xs -> if null xs then [] else x : rec tail xs)
```

1.2 sublistaQueMásSuma

```
sublistaQueMasSuma :: [ Int ] -> [Int]
sublistaQueMasSuma =
recr (\x xs res ->
    if ( sum . prefijoQueM   a   sSuma ) ( x : xs ) >= sum res
    then prefijoQueM   a   sSuma ( x : xs )
    else res
    ) []
```

Como necesitamos acceder en cada paso a la **subestructura (el resto de la lista) utilizamos recursión primitiva**. O sea, estamos usando `xs` en algo que no es el llamado recursivo.

1.3 Generación infinita

pares

Queremos una lista infinita que contenga todos los pares de números naturales sin repetir :

```
pares :: [(Int, Int)]
pares = [(x,y) | x <- [0..], y <- [0..]]
```

En este escenario sólo se generan pares con `x = 0`. La idea para que funcione la generación infinita es poder decir en qué posición está cierto par (noción de orden). Como se ve en este caso :

```
pares :: [( Int , Int) ]
pares = [ p | k <- [0..], p <- paresQueSuman k]

paresQueSuman :: Int -> [( Int , Int) ]
paresQueSuman k = [(i, k-i) | i <- [0..k]]
```

En este escenario, aparece un poco la idea de orden, si tuviésemos memoria infinita, podríamos encontrar la posición del (2,1).

2 Folds sobre estructuras nuevas

2.1 AEB

Se define el siguiente tipo

```
data AEB a = Hoja a | Bin (AEB a) a (AEB a)

miAbol = Bin (Hoja 3) 5 (Bin (Hoja 7) 8 (Hoja 1))
```

Un árbol estrictamentente binario, no puede tener un hijo de un lado y no del otro.

Definir el esquema de recursión estructural (fold) y dar su tipo

Para lograrlo, primero miremos el tipo de `foldr`, el esquema de recursión estructural para listas.

$$foldr :: (a \to b \to b) \to b \to [a] \to b$$

Si pensamos en por qué tiene ese tipo y en particular en cuáles son los constructores de `[a]`, sabemos que **hay un argumento por cada constructor, y luego le entra una lista y devuelve un resultado**.
Un esquema de recursión estructural espera **recibir un argumento por cada constructor**(para saber qué devolver en cada caso), y además **la estructura que va a recorrer.** El tipo de cada agumento va a depender de lo que reciba el constructor correspondiente. *Si el constructor es recursivo, el argumnento correspondiente del fold va a recibir el resultado de cada llamada recursiva.*
Miremos entonces la estructura del tipo

```
data AEB a = Hoja a | Bin (AEB a) a (AEB a)
```

Estamos frente a un tipo inductivo con un constructor **no recursivo** y uno **recursivo**. Por tanto, el tipo de nuestro `fold`

```
foldAEB ::  (a -> b) -> (b -> a -> b -> b) -> AEB a -> b
```

En naranja vemos el resultado de la recursión sobre cada `AEB` que forma parte del constructor `Bin (AEB a) a (AEB a)`. Si por ejemplo hubiese habido un `AEB a = Nil` entonces entraría `fNil ::  b`.

```
aeb :: AEB Int
aeb = Bin (Hoja 3) 5 (Bin (Hoja 7) 8 (Hoja 1))

foldAEB :: (a -> b) -> (b -> a -> b -> b) -> AEB a -> b
foldAEB fHoja fBin t = case t of
    Hoja n      -> fHoja n
    Bin t1 n t2 -> fBin (rec t1) n (rec t2)
  where
    rec = foldAEB fHoja fBin
```

Y ahora con este esquema definir las siguientes funciones. Ojo con lo que le pasamos a cada función como caso base. Tiene que se una función!

```haskell
alturaAEB :: AEB a -> Int
alturaAEB = foldAEB (const 1) (\recI _ recD -> 1 + max recI recD)

ramasAEB :: AEB a -> [[a]]
ramasAEB = foldAEB (\x -> [[x]]) (\recI r recD -> map (r:) (recI ++ recD))

cantNodosAEB :: AEB a -> Int
cantNodosAEB = foldAEB (const 1) (\recI _ recD -> 1 + recI + recD)

cantHojasAEB :: AEB a -> Int
cantHojasAEB = foldAEB (const 1) (\recI _ recD -> recI + recD)

-- Recuerden que los constructores tambi n son funciones
espejoAEB :: AEB a -> AEB a
espejoAEB = foldAEB Hoja (\recI r recD -> Bin recD r recI)
```

2.2 `AB`

Dado el siguiente tipo de datos :

```haskell
data AB a = Nil | Bin (AB a) a (AB a)
```

Qué tipo de recursión tiene cada una de las siguientes funciones ?

La recursión **global puede acceder a los resultados de recursiones anteriores, no sólo a la última**.

- `insertarABB`

```haskell
insertarABB :: Ord a => a -> AB a -> AB a
insertarABB x Nil = Bin Nil x Nil
insertarABB x (Bin i r d) =
            if x < r
            then Bin (insertarABB x i) r d
            else Bin i r (insertarABB x d)
```

En este caso, la recursión es **primitiva** ya que accedemos a `i` y a `d` sin estar dentro del llamado. Escrito con su esquema correspondiente :

```haskell
insertarABB x = recABB (Bin Nil x Nil) (\i r d recI recD -> if x < r then Bin recI r d else Bin i r recD)
```

`truncar`

```haskell
truncar :: AB a -> Int -> AB a
truncar Nil _ = Nil
truncar (Bin i r d) n =
        if n == 0
        then Nil
        else Bin (truncar i (n-1)) r (truncar d (n-1))
```

En este caso, la recursión es estructural, ya que sólo usamos la subestructura como argumento de truncar, y no se le pasa a truncar la estructura entera. Escrito con su esquema correspondiente :

```haskell
truncar = foldABB (const Nil) (\recI r recD -> \n -> if n == 0 then Nil else Bin recI r recD)
```

2.3 `Polinomio`

Se define el siguiente tipo que representa polinomios :

```haskell
data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)
```

Nos piden definir la funcióon `evaluar`, el esquema de recursión estructural `foldPoli` y reescribir `evaluar` usando `foldPoli`

```haskell
foldPoli :: b -> (a -> b) -> (b -> b -> b) -> (b -> b -> b) -> Polinomio a -> b
foldPoli cX cCte cSuma cProd p = case p of
    X         -> cX
    Cte k     -> cCte k
    Suma p’ q -> cSuma (rec p’) (rec q)
    Prod p’ q -> cProd (rec p’) (rec q)
  where
    rec = foldPoli cX cCte cSuma cProd
```

```haskell
-- Usa case of en vez de pattern matching. Es lo mismo pero case of escribe menos y podes usar un solo where porque es ua sola ecuaci n

evaluar :: Num a => a -> Polinomio a -> a
{-
Evaluar e p = case e of
        x -> e
        Cte i -> i
        Suma p' q -> (evaluar e p') + (evaluar e q)
        Prod p' q -> (evaluar e p') * (evaluar e q)
-}
evaluar e = foldPoli e id (+) (*)
```

2.4 RoseTree

Se define el tipo de datos

```haskell
data RoseTree a = Rose a [RoseTree a]
```

de árboles donde cada nodo tiene una **cantidad indeterminada de hijos**. Nos piden escribir el esquema de recursión estructural para el tipo y 4 funciones (`hojas, ramas, tamaño, altura`). Importante : **rec en foldRose es una lista de resultados.**

```haskell
rose = Rose 2 [Rose 3 [], Rose 4 [Rose 5 []]]

foldRose :: (a -> [b] -> b) -> RoseTree a -> b
foldRose f (Rose r rs) = f r (map (foldRose f) rs)
-- Con map (map (foldRose f) rs). A cada RoseTree de rs le aplica la funci n foldrRose f. As  obtiene [b]. Luego combina eso con r usando f.

hojasRose :: RoseTree a -> [a]
hojasRose = foldRose (\r rec -> if null rec
                                then [r]
                                else concat rec)

ramasRose :: RoseTree a -> [[a]]
ramasRose = foldRose (\r rec -> if null rec
                                then [[r]]
                                else map (r:) (concat rec))

tama oRose :: RoseTree a -> Int
tama oRose = foldRose (\_ rec -> 1 + sum rec)

alturaRose :: RoseTree a -> Int
alturaRose = foldRose (\_ rec -> if null rec
                                 then 1
                                 else 1 + maximum rec)
```

3 Funciones como estructuras de datos

Se cuenta con la siguiente representación de conjuntos

```
type Conj a = (a -> Bool)
```

caracterizados por su función de pertenencia. De este modo si c es un conjunto y e un elemento, la expresión $c\ e$ devuelve `True` si $e \in c$ y `False` en caso contrario.

Nos piden definir la constante vacío y las funciones `intersección, unión , diferencia`.

Detalles : `type` es un alias (sinónimo de tipo) o sea, renombra algo que ya existe, **no crea nuevo tipo como** `Data`. `newType` Crea un nuevo tipo que envuelve exactamente un valor. Es un tipo distinto a nivel de tipos. Se usa para seguridad de tipos, instancias separadas, ...

```haskell
type Conj a = (a -> Bool)

vacio :: Conj a
vacio = const False
-- Define el conjunto vac o con una funci n que dice si un elemento esta o no. vac o elem = False para todo elem

-- agregar :: Eq a => a -> (a -> Bool) -> (a -> Bool)
agregar :: Eq a => a -> Conj a -> Conj a
agregar e c = \x -> x == e || c x
```

```haskell

union :: Conj a -> Conj a -> Conj a
union c1 c2 = \x -> c1 x || c2 x

interseccion :: Conj a -> Conj a -> Conj a
interseccion c1 c2 = \x -> c1 x && c2 x

diferencia :: Conj a -> Conj a -> Conj a
diferencia c1 c2 = \x -> c1 x && not (c2 x)
```