Prácticas: Paradigmas de Lenguajes de Programación

Zamboni, Gianfranco

5 de febrero de 2018

Índice

0. Práctica 0

0.1. Ejercicio 1

`null :: Foldable t => t a -> Bool` indica si una estructura está vacía. El tipo a debe ser de la clase Foldable, esto es, son tipos a los que se les puede aplicar la función foldr. La notación "t aïndica que es un tipo parámetrico, es decir, un tipo t que usa a otro tipo a, por ejemplo, si le pasamos a la función una lista de enteros, entonces a = Int y t = [Int]

`head :: [a] -> a` devuelve el primer elemento de una lista.

`tail :: [a] -> [a]` devuelve los últimos elementos de una lista (todos los elementos, salvo el primero).

`init :: [a] -> [a]` devuelve los primeros elementos de una lista (todos los elementos salvo el último).

`last :: [a] -> a` devuelve el último elemento de una lista.

`take :: Int -> [a] -> [a]` devuelve los primeros n elementos de una lista

`drop :: Int -> [a] -> [a]` devuelve los últimos n elementos de una lista

`(++) :: [a] -> [a] -> [a]` concatena dos listas

`concat :: Foldable t => t [a] -> [a]` concatena todas las listas de un contenedor de listas que soporte la operación foldr.

`(!!) :: [a] -> Int -> a` devuelve el elemento de una lista `l` que se encuentra en la `n`-ésima posición. La numeración comienza desde 0.

`elem :: (Eq a, Foldable t) => a -> t a -> Bool`: Dada una estructura T que soporta la operación foldr y que almacene elementos del tipo a que puedan ser comparados por medio de la igualdad y dado un elemento A de ese tipo, indica si A aparecen en T.

0.2. Ejercicio 2

```haskell
-- La función abs de Prelude ya hace esto
valorAbsoluto :: Float -> Float
valorAbsoluto x | x < 0     = -x
                | otherwise =  x

bisiesto :: Int -> Bool
bisiesto x = (x `mod` 4) == 0

factorial :: Int -> Int
factorial 1 = 1
factorial x = x * factorial (x-1)

cantDivisoresPrimos :: Int -> Int
cantDivisoresPrimos x = length (filter esPrimo (divisores x))
```

```haskell
-- Auxiliares

esPrimo :: Int -> Bool
esPrimo x = length (divisores x) == 2

divisores :: Int -> [Int]
divisores x = [ y | y <- [1..x], x `mod` y == 0 ];
```

0.3. Ejercicio 3

```haskell
inverso :: Float -> Maybe Float
inverso 0 = Nothing
inverso x = Just (1/x)

aEntero :: Either Int Bool -> Int
aEntero (Left x) = x
aEntero (Right x) | x == True = 1
                  | otherwise = 0
```

0.4. Ejercicio 4

```haskell
limpiar :: String -> String -> String
limpiar xs ys = [ y | y <- ys, not(elem y xs) ]

difPromedio :: [Float] -> [Float]
difPromedio xs = map (\y -> y - promedio xs) xs
    where promedio xs = (sum xs) / (genericLength xs)

todosIguales :: [Int] -> Bool
todosIguales =
        foldr (\y  rec -> ((length xs == 1) || (y == (head xs)))
                && rec) True
```

0.5. Ejercicio 5

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

vacioAB:: AB a -> Bool
vacioAB Nil = True
vacioAB (Bin _ _ _) = False

negacionAB :: AB Bool -> AB Bool
negacionAB Nil = Nil
negacionAB (Bin l x r) =
  Bin (negacionAB l) (not x) (negacionAB r)

productoAB :: AB Int -> Int
productoAB Nil = 1
productoAB (Bin l x r) = x * (productoAB l) * (productoAB r)
```

1. Práctica 1

1.1. Ejercicio 1

```haskell
-- La función max de Prelude ya hace esto
max2 ::(Float, Float) -> Float
max2 (x, y) | x >= y = x
            | otherwise = y

max2Currificada :: Float -> Float -> Float
max2Currificada x y | x >= y = x
         | otherwise = y

normaVectorial :: (Float, Float) -> Float
normaVectorial (x, y) = sqrt (x^2 + y^2)

normaVectorial :: Float -> Float -> Float
normaVectorial x y = sqrt (x^2 + y^2)

-- subtract ys esta definida en Prelude
subtract1 :: Float -> Float -> Float
subtract1 = flip (-)

-- La función pred definida en Prelude ya hace esto
predecesor :: Float -> Float
predecesor = subtract 1

evaluarEnCero :: (Float -> b) -> b
evaluarEnCero = \f -> f 0

dosVeces :: (a -> a) -> (a -> a)
dosVeces = \f -> f.f

flipAll :: [a -> b -> c] -> [ b -> a -> c]
flipAll = map flip

flipRaro :: b -> ( a -> b -> c ) -> a -> c
flipRaro = flip flip
```

1.2. Ejercicio 2

```haskell
[ x | x <- [1..3], y <- [x..3], ( x + y ) `mod` 3 == 0 ]
  = [ 1, 3 ]
```

1.3. Ejercicio 3

```
pitagoricas :: [(Integer, Integer, Integer)]
pitagoricas =[(a, b, c) | a <- [1..], b <-[1..], c <- [1..],
              a^2 + b^2 == c^2]
```

Esta definición agrega la tupla `(1,1,1)` a la lista y luego aumenta `c` infinitamente, sin encontrar ningun nueva coincidencia. Si cambiamos el orden en el que se recorren las listas y agregando algunas cotas de la siguiente forma:

```
pitagoricas :: [(Integer, Integer, Integer)]
pitagoricas = [ (a, b, c) | c <- [1..], b <-[1..c], a <- [1..c],
                a^2 + b^2 == c^2]
```

En este caso, para cada número probamos todas las combinaciones de pares `(a,b)` tales que la suma de sus cuadrados podría llegar a dar `c`. Como `a` y `b` están acotados por `c`, ya que claramante `c^2 + c^2 > c^2`, la cantidad de pruebas de pares para cada número es finita (2^c pares) y es posible pasar al siguiente número una vez realizados estos chequeos.

1.4. Ejercicio 4

```
primerosPrimos :: Int -> [Int]
primerosPrimos n = take n [ x | x <- [2..], esPrimo x ]
```

Gracias a la evaluación *lazy*, cuando se encuentran los primeros `n` primos la función deja de computar la lista de primos.

1.5. Ejercicio 5

```
partir :: [a] -> [ ([a], [a]) ]
partir xs = [ (take i xs, drop i xs) | i <- [0..(length xs)] ]
```

1.6. Ejercicio 6

```
listasQueSuman :: Int -> [[Int]]
listasQueSuman 1 = [[1]]
listasQueSuman n = [n]:( concat
  [ map ( (n-i): ) ( listasQueSuman i ) | i <- [ 1..n-1 ] ] )
```

A preguntar: La función definiida usa recursión explicita. Podemos definir el esquema recursivo `indcucciónGlobal` y volver a definir listasQueSuman usando este esquema:

```
induccionGlobal:: (Int -> Int -> b -> b) ->
  (Int -> [b] -> b) -> b -> Int -> b
induccionGlobal _ _ z 1 = z
induccionGlobal g f z n =
  f n [ g n i ( induccionGlobal g f z i ) | i <- [ 1..(n-1) ] ]

listasQueSumanIG :: Int -> [[Int]]
listasQueSumanIG n = induccionGlobal
(\i j -> map ((i-j):) )
(\x xs -> [x]:(concat xs))
[[1]] n
```

El esquema `induccionGlobal` nos permite definir funciones recursivas en las que el resultado de la función depende de todos los casos anteriores, no solo del caso anterios.

1.7. Ejercicio 7

```haskell
listasFinitas :: [[Int]]
listasFinitas = concat [ listasQueSuman i | i <- [1..]]
```

1.8. Ejercicio 8

```haskell
-- curry y uncurry ya están definidas en Prelude
curry1 :: ((a,b) -> c) -> a -> b -> c
curry1 f a b = f (a,b)

uncurry1 :: (a -> b -> c) -> (a, b) -> c
uncurry1 f (a, b) = f a b
```

A preguntar: No podemos definir una función `curryN` que tome una función con un número arbitrario de parametros, ya que la cantidad de parámetros de la función currificada depende de la cantidad de parámetros de la función original. Esto significa que `curryN` debería poder modificar la cantidad de parámetros que toma dependiendo de la función que se le pasa, lo que es imposible.

Otra idea sería tratar de definirla de manera que dada una función vaya remplazando los parámetros de a poco generando, de esta forma, n funciones parciales. Pero esto es imposible ya que la función debe tener la tupla de parámetros completa para poder ser evaluada de cualquier manera.

1.9. Ejercicio 9

```haskell
dc :: DivideConquer a b
dc esTrivial resolver repartir combinar x =
  if esTrivial x then
    resolver x
  else combinar (map  dc1 (repartir x))
  where dc1 = dc esTrivial resolver repartir combinar

mergesort :: Ord a => [a] -> [a]
mergesort = dc ((<=1).length)
  id
  partirALaMitad
  (\[xs,ys] -> merge xs ys)

mapD :: (a -> b) -> [a] -> [b]
mapDC f = dc ((<=1).length)
  ( \xs -> if (length xs) == 0 then [] else [ f (head xs) ] )
  partirALaMitad
  concat

filterDC :: (a -> Bool) -> [a] -> [a]
filterDC p = dc ((<=1).length)
  (\xs -> if (length xs == 0) || (p (head xs)) then [] else xs )
  partirALaMitad
  concat
```

```haskell
-- Auxiliares

partirALaMitad :: [a] -> [[a]]
partirALaMitad xs = [ take i xs, drop i xs ]
  where i = (div (length xs) 2)

merge :: Ord a => [a] -> [a] -> [a]
merge = foldr
  (\y rec -> (filter (<= y) rec) ++ [y] ++ (filter (>y) rec))
```

1.10. Ejercicio 10

```haskell
sumFold :: Num a => [a] -> a
sumFold = foldr (+) 0

elemFold :: Eq a => a -> [a] -> Bool
elemFold x = foldr (\y rec -> (y==x) || rec) False

masMasFold :: [a] -> [a] -> [a]
masMasFold = flip (foldr (\x rec-> x:rec) )

mapFold :: (a->b) -> [a] -> [b]
mapFold f = foldr (\x rec-> (f x):rec) []

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

La función `foldr1 :: Foldable t => (a -> a -> a) -> t a -> a` está definida en Prelude. Esta función es una variante de `foldr` en la que el caso base se da cuando la estructura contiene un único elemento y ese elemento es el resultado del caso base.

```haskell
mejorSegun :: (a -> a -> Bool) -> [a] -> a
mejorSegun f xs =
  foldr1 (\x rec -> if f x rec then x else rec) xs
```