Programación Funcional en Haskell

Paradigmas de Lenguajes de Programación

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

31 de Enero de 2018

Hoy presentamos...

1. Ejercicios sobre listas

2. Folds sobre listas
   - FoldR
   - FoldL

Ejercicios sobre listas

Definir sin utilizar recursión explicita

- `reverseAnidado :: [[Char]] -> [[Char]]` que, dada una lista de strings, devuelve una lista con cada string dado vuelta y la lista completa dada vuelta. Ej: `reverseAnidado ["hola", "que", "tal"]` $\rightsquigarrow$ `["lat", "euq", "aloh"]`
- `shuffle :: [Int] -> [a] -> [a]` que, dada una lista de índices $[i_1, ..., i_n]$ y una lista l, devuelve la lista $[l_{i_1}, ..., l_{i_n}]$.
- `soloPuntosFijos :: [Int -> Int] -> Int -> [Int -> Int]` que toma una lista de funciones y un número n. En el resultado, deja las funciones que al aplicarlas a n dan n.
- `paresCuadrados :: [Int] -> [Int]` que, dada una lista de enteros, devuelve una lista con los cuadrados de los números pares.

Esquemas de recursión sobre listas: FoldR

FoldR

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

La función `foldr` nos permite realizar recursión estructural sobre una lista.

O, dicho de otra forma, la función `foldr`

- Toma una función que representa el paso recursivo y un valor que representa el caso base,
- Y nos devuelve una función que sabe como reducir listas de **a** a un valor **b**.

Esquemas de recursión sobre listas: FoldR

FoldR

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

¿Cómo funciona?

```
suma xs = foldr (+) 0 xs
> suma [1,2,3]
---> foldr (+) 0 [1,2,3]
---> 1 + (foldr (+) 0 [2,3])
---> 1 + (2 + (foldr (+) 0 [3]))
---> 1 + (2 + (3 + (foldr (+) 0 [])))
---> 1 + (2 + (3 + 0))
---> 1 + (2 + 3)
---> 1 + 5
---> 6
```

Notar que el primer `(+)` que se puede resolver es entre el último elemento de la lista y el caso base del `foldr`. Por esta razón decimos que el `foldr` *acumula* el resultado desde la **derecha**.

Esquemas de recursión sobre listas: FoldR

FoldR

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

Definir utilizando foldr

- `producto :: [Int] -> Int`
- `concatenar :: [[a]] -> [a]`
- `map :: (a -> b) -> [a] -> [b]`
- `filter :: (a -> Bool) ->[a] -> [a]`

Esquemas de recursión sobre listas: FoldL

La función `foldl` es muy similar a `foldr` pero *acumula* desde la **izquierda**.
Se define de la siguiente forma:

FoldL

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

¿Cómo funciona?

```
suma xs = foldl (+) 0 xs
> suma [1,2,3]
---> foldl (+) 0 [1,2,3]
---> foldl (+) (0 + 1) [2,3]
---> foldl (+) ((0 + 1) + 2) [3]
---> foldl (+) (((0 + 1) + 2) + 3) []
---> (((0 + 1) + 2) + 3)
---> ((1 + 2) + 3)
---> (3 + 3)
---> 6
```

Notar que el primer `(+)` que se puede resolver es entre el primer elemento de la lista y el caso base del `foldl`.

Esquemas de recursión sobre listas: FoldL

FoldL

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

Definir utilizando foldl

- `producto :: [Int] -> Int`
- `reverso :: [a] -> [a]`

Esquemas de recursión sobre listas: FoldR, FoldL y las listas infinitas

¿Qué sucede con las listas infinitas al usar `foldr` o `foldl`?

Usando foldr

```
suma [1..]
---> foldr (+) 0 [1..]
---> 1 + (foldr (+) 0 [2..])
---> 1 + (2 + (foldr (+) 0 [3..]))
---> 1 + (2 + (3 + (foldr (+) 0 [4..])))
```

Usando foldl

```
suma [1..]
---> foldl (+) 0 [1..]
---> foldl (+) (0 + 1) [2..]
---> foldl (+) ((0 + 1) + 2) [3..]
---> foldl (+) (((0 + 1) + 2) + 3) [4..]
```

Esquemas de recursión sobre listas: FoldR1 y FoldL1

Para situaciones en las cuales no hay un caso base claro (ej: no existe el neutro), tenemos las funciones: `foldr1` y `foldl1`. Permiten hacer recursión estructural sobre listas sin definir un caso base:

- `foldr1` toma como caso base el último elemento de la lista.
- `foldl1` toma como caso base el primer elemento de la lista.

Para ambas, la lista **no** debe ser vacía.

Definir las siguientes funciones

- `ultimo :: [a] -> a`
- `maximum :: Ord a => [a] -> a`

¡Las difíciles!

Sin usar recursión explícita:

```
pertenece :: Eq a => a -> [a] -> Bool
pertenece e = foldr ...
```

Definir la función take, ¿cuál es la diferencia?

```
take :: Int -> [a] -> [a]
take n = foldr ...
```

Fin (por ahora)

¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿?