Programación Funcional en Haskell

Paradigmas de Lenguajes de Programación

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

21 de agosto de 2018

Generación Infinita

Ejercicio: definir las siguientes funciones

- `pares :: [(Int, Int)]`, una lista (infinita) que contenga todos los pares de números naturales (sin repetir).
- `triplas :: [(Int, Int, Int)]`, una lista (infinita) que contenga todas las triplas de números naturales (sin repetir).

Un poco más difícil

```
listasQueSuman :: Int -> [[Int]]
```

que, dado un número natural n, devuelve todas las listas de enteros mayores o iguales que 1 cuya suma sea n

```
listasPositivas :: [[Int]]
```

que contenga todas las listas finitas de enteros mayores o iguales que 1.

Volvemos a las listas

Para resolver en el aire

Definir las siguientes funciones sin usar recursión explícita:

- `negar :: [[Char]] -> [[Char]]`, que, dada una lista de palabras, le agrega "in" adelante a todas. Por ejemplo `negar [``util'', ``creible''] ⇝ [``inutil'', ``increible'']`
- `sinVacías :: [[a]] -> [[a]]`, que, dada una lista de listas, devuelve las que no son vacías (en el mismo orden).

Ahora sí

Definir las siguientes funciones:

- `all :: (a -> Bool) -> [a] -> Bool`, que decide si todos los elementos de una lista cumplen una cierta propiedad.
- `concat :: [[a]] -> [a]`, que dada una lista de listas, devuelve la lista que resulta de concatenarlas en orden.

¿Qué esquema de recursión podemos usar en estos casos?

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

- `longitud :: [a] -> Int`
- `producto :: [Int] -> Int`
- `concat :: [[a]] -> [a]`
- `all :: (a -> Bool) -> [a] -> Bool`
- `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: FoldR, FoldL y las listas infinitas

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

Usando foldr

```
all even [0..]
---> foldr (\x r -> even x && r) True [0..]
---> even 0 && (foldr (\x r -> even x && r) True [1..])
---> True && (foldr (\x r -> even x && r) True [1..])
---> foldr (\x r -> even x && r) True [1..]
---> even 1 && (foldr (\x r -> even p x && r) True [2..])
---> False && (foldr (\x r -> even x && r) True [2..])
---> False
```

Usando foldl

```
all even [0..]
---> foldl (\a x -> even x && a) True [0..]
-->* foldl (\a x -> even x && a) (even 0 && True) [1..]
-->* foldl (\a x -> even x && a) (even 1 && (even 0 && True)) [2..]
-->* foldl (...) (even 2 && (even 1 && (even 0 && True))) [3..]
-->* foldl (...) (even 3 && (even 2 && (...))) [4..]
```

*: No es *exactamente* el orden en el que reduciría Haskell (¿por qué?) pero el ejemplo vale igual.

Esquemas de recursión estructural sobre listas

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

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

Esquemas de recursión estructural sobre listas

¿Qué computan estas funciones?

- ```
  f1 :: [Bool] -> Bool
  f1 = foldr (&&) True
  ```
- ```
  f2 :: [a] -> [a]
  f2 = foldr (:) []
  ```
- ```
  f3 :: [a] -> [a] -> [a]
  f3 xs ys = foldr (:) ys xs
  ```
- ```
  f4 :: [a] -> [a]
  f4 = foldl (flip (:)) []
  ```

¿Se puede escribir la función `insertarOrdenado :: Ord a => a -> [a] -> [a]` usando `foldr`?

Recursión primitiva

FoldR

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

¿Cómo se puede escribir la función
`insertarOrdenado :: Ord a => a -> [a] -> [a]` usando `recr`?

¡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 ...
```

Tipos algebraicos y su definición en Haskell

Tipos algebraicos

- definidos como **combinación de otros tipos**
- están formados por uno o más constructores
- cada constructor puede o no tener argumentos
- los argumentos de los constructores pueden ser recursivos
- se inspeccionan usando *pattern matching*
- se definen mediante la cláusula `data`

Algunos ejemplos

```
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b

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

Folds sobre estructuras nuevas

Sea el siguiente tipo:

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

Ejemplo: `miÁrbol = Bin (Hoja 3) 5 (Bin (Hoja 7) 8 (Hoja 1))`

Definir el esquema de recursión estructural (*fold*) para árboles binarios, y dar su tipo.

El esquema debe permitir definir las funciones `altura`, `ramas`, `#nodos`, `#hojas`, `espejo`, etc.

¿Cómo hacemos?

Recordemos el tipo de `foldr`, el esquema de recursión estructural para listas.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

¿Por qué tiene ese tipo?
(Pista: pensar en cuáles son los constructores del tipo `[a]`).

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 argumento va a depender de lo que reciba el constructor correspondiente. (¡Y todos van a devolver lo mismo!)

Si el constructor es recursivo, el argumento correspondiente del fold va a recibir el resultado de cada llamada recursiva.

¿Cómo hacemos? (Continúa)

Miremos bien la estructura del tipo.

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

Estamos ante un tipo inductivo con un constructor *no recursivo* y un constructor *recursivo*.

¿Cuál va a ser el tipo de nuestro `fold`?

¿Y la implementación?

Solución

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

Ejercicio para ustedes: definir las funciones `altura`, `ramas`, `#nodos`, `#hojas` y `espejo` usando `foldAB`.

Si quieren podemos hacer alguna en el pizarrón.

Folds sobre estructuras nuevas

Definir el esquema de recursión estructural para el siguiente tipo:

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

Ejercicio

Usando el esquema definido, escribir la función
`evaluar :: Num a => a -> Polinomio a -> a`.

Solución

```
foldPoli::Num a=>b->(a->b)->(b->b->b)->(b->b->b)->Polinomio a->b
foldPoli casoX casoCte casoSuma casoProd pol = case pol of
        X -> casoX
        Cte n -> casoCte n
        Suma p q -> casoSuma (rec p) (rec q)
        Prod p q -> casoProd (rec p) (rec q)
        where rec = foldPoli casoX casoCte casoSuma casoProd


evaluar::Num a=>a->Polinomio a->a
evaluar n = foldPoli n id (+) (*)
```

Algo un poco más complejo... ¿o no?

Sea el tipo de datos `RoseTree` de árboles no vacíos, donde cada nodo tiene una cantidad indeterminada de hijos.

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

1. Escribir el esquema de recursión estructural para `RoseTree`. Es importante escribir primero su tipo.
2. Usando el esquema definido, escribir las siguientes funciones:
   1. **hojas**, que dado un `RoseTree`, devuelve una lista con sus hojas ordenadas de izquierda a derecha, según su aparición en el `RoseTree`.
   2. **distancias**, que dado un `RoseTree`, devuelve las distancias de su raíz a cada una de sus hojas.
   3. **altura**, que devuelve la altura de un `RoseTree` (la cantidad de nodos de la rama más larga). Si el `RoseTree` es una hoja, se considera que su altura es 1.

Tipo RoseTree

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

¿Cómo hacemos el fold?

¿Hay caso/s base?

¿Cuántos llamados recursivos hay que hacer?

Empecemos por el tipo

Recordar: un `fold` siempre toma un argumento por cada constructor del tipo, y además la estructura que va a recorrer.

Los argumentos que corresponden a los constructores devuelven siempre algo del tipo que queremos devolver, y reciben tantos argumentos comos sus respectivos constructores.

Si el constructor es recursivo, el argumento correspondiente del fold va a recibir el resultado de cada llamada recursiva.

¿Cómo se aplica esto al RoseTree?

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

En el caso del `RoseTree`, eso no cambia. Si nuestro constructor toma una lista de RoseTrees, tendremos una lista de resultados de las recursiones.

Entonces el tipo del `fold` es...

```
foldRT::(a->[b]->b)->RoseTree a->b
```

Ahora implementémoslo

```
foldRT::(a->[b]->b)->RoseTree a->b
foldRT f (Rose x hs) = f x (map (foldRT f) hs)
```

Y finalmente...

1. Definir el tipo de datos `RoseTree` de árboles no vacíos, donde cada nodo tiene una cantidad indeterminada de hijos. ✓
2. Escribir el esquema de recursión estructural para `RoseTree`. Es importante escribir primero su tipo. ✓
3. Usando el esquema definido, escribir las siguientes funciones:
   1. **hojas**, que dado un `RoseTree`, devuelve una lista con sus hojas ordenadas de izquierda a derecha, según su aparición en el `RoseTree`.
   2. **distancias**, que dado un `RoseTree`, devuelve las distancias de su raíz a cada una de sus hojas.
   3. **altura**, que devuelve la altura de un `RoseTree` (la cantidad de nodos de la rama más larga). Si el `RoseTree` es una hoja, se considera que su altura es 1.

Ejercicio

Se cuenta con la siguiente representación de conjuntos, caracterizados por su función de pertenencia:

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

De este modo, si `conj1` es un conjunto y `e` un elemento, la expresión `conj1 e` devuelve `True` si `e` pertenece a `conj1`, y `False` en caso contrario.

Operaciones sobre conjuntos

- Definir y dar el tipo de las siguientes funciones:
  - vacío
  - unión
  - intersección
  - complemento
  - singleton
- ¿Puede definirse un map para esta estructura?
  Para utilizar, por ejemplo, de esta manera: `(mapC (+1) conj1) e`
- ¿Puede definirse la función `esVacío :: Conj a -> Bool`?
  ¿Y `esVacío :: Conj Bool -> Bool`?
- *Si $A \subseteq \mathbb{N}$ es computable, entonces existe una enumeración computable estrictamente creciente de los elementos de A.*[a]
  Demostrar esta afirmación programando la susodicha enumeración.

[a] Existe una $f : \mathbb{N} \to \mathbb{N}$ computable y estrictamente creciente tal que $A = \{f(0), f(1), f(2), \ldots\}$

Fin (por ahora)

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