Programación Funcional (parte 2)

Otras estructuras de datos

Paradigmas de (lenguajes de) programación

Departamento de Computación
FCEyN UBA

1 de abril de 2025

Un breve repaso

Ya conocemos `foldr` y `foldl`.

Un breve repaso

Ya conocemos `foldr` y `foldl`. Intentemos definir la función `maximoL` que tiene como precondición que la lista no sea vacía.

```
1 maximoL :: Ord a => [a] -> a
```

Utilizando `max`, definida en el preludio

```
1 max :: (Ord a) => a -> a -> a
```

Un breve repaso

Veamos una posible definición

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

Un breve repaso

Veamos una posible definición

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

Notemos que la precondición nos garantiza poder aplicar el siguiente patrón:

```haskell
-- Pre: lista no vacía
maximoL :: (Ord a) => [a] -> a
maximoL (x:xs) = foldr max x xs
```

Un breve repaso

Dentro de la familia de folds sobre listas existen algunas funciones adicionales, por ejemplo foldr1

```
1 -- Pre: lista no vacía
2 foldr1 :: (a -> a -> a) -> [a] -> a
```

Un breve repaso

Dentro de la familia de folds sobre listas existen algunas funciones adicionales, por ejemplo foldr1

```
1 -- Pre: lista no vacía
2 foldr1 :: (a -> a -> a) -> [a] -> a
```

Con esta función podriamos escribir `maximoL` de la siguiente manera

```
1 maximoL = foldr1 max
```

Un breve repaso

Las variantes de `foldr` abstraen el esquema de recursión estructural.

Un breve repaso

Las variantes de `foldr` abstraen el esquema de recursión estructural.
Pero, ¿Y si no está hecha con `foldr`?

```
1 take' :: [a] -> Int -> [a]
2 take' []     _ = []
3 take' (x:xs) n = if n == 0
4                  then []
5                  else x : take' xs (n-1)
```

Un breve repaso

Las variantes de `foldr` abstraen el esquema de recursión estructural.
Pero, ¿Y si no está hecha con `foldr`?

```
1 take' :: [a] -> Int -> [a]
2 take' []     _ = []
3 take' (x:xs) n = if n == 0
4                  then []
5                  else x : take' xs (n-1)
```

Es esctructural, ya que se usa el argumento inductivo de la lista (la cola).

Un breve repaso

Ahora, queremos escribir la función `sublistaQueMásSuma :: [Int] -> [Int]`.

Un breve repaso

Ahora, queremos escribir la función `sublistaQueMásSuma :: [Int] -> [Int]`.

```
1 sublistaQueMásSuma :: [Int] -> [Int]
2 sublistaQueMásSuma =
3     recr (\x xs res ->
4             if (sum . prefijoQueMásSuma) (x:xs) >= sum res
5             then prefijoQueMásSuma (x:xs)
6             else res
7           ) []
```

Un breve repaso

Ahora, queremos escribir la función `sublistaQueMásSuma :: [Int] -> [Int]`.

```
1 sublistaQueMásSuma :: [Int] -> [Int]
2 sublistaQueMásSuma =
3     recr (\x xs res ->
4             if (sum . prefijoQueMásSuma) (x:xs) >= sum res
5             then prefijoQueMásSuma (x:xs)
6             else res
7           ) []
```

Como necesitamos acceder en cada paso a la subestructura (el resto de la lista), utilizamos recursión primitiva.

Repaso - Generación Infinita

Definir:

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

Repaso - Generación Infinita

Definir:

`pares :: [(Int, Int)]`, una lista (infinita) que contenga todos los pares de números naturales (sin repetir). Veamos una opción:

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

Repaso - Generación Infinita

Definir:

`pares :: [(Int, Int)]`, una lista (infinita) que contenga todos los pares de números naturales (sin repetir). Veamos una opción:

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

¿Y en qué posición está el `(2,1)`?¿Se genera?

Repaso - Generación Infinita

Definir:

`pares :: [(Int, Int)]`, una lista (infinita) que contenga todos los pares de números naturales (sin repetir). Veamos una opción:

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

¿Y en qué posición está el `(2,1)`?¿Se genera?

```
1 pares :: [(Int, Int)]
2 pares  = [ p | k <- [0..], p <- paresQueSuman k]
3
4 paresQueSuman :: Int -> [(Int, Int)]
5 paresQueSuman k = [(i, k-i) | i <- [0..k]]
```

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 estrictamente binarios, y dar su tipo.

El esquema debe permitir definir las funciones `altura`, `ramas`, `cantNodos`, `cantHojas`, `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]`).

¿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*.

¿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`?

¿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

```haskell
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
```

Ejercicio para ustedes: definir las funciones `altura`, `ramas`, `cantNodos`, `cantHojas` y `espejo` usando `foldAEB`.

Si quieren podemos hacer alguna en el pizarrón.

Funciones sobre árboles

Dado el tipo de datos:

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

¿Qué tipo de recursión tiene cada una de las siguientes funciones? (Estructural, primitiva, global).

```
1 insertarABB :: Ord a => a -> AB a -> AB a
2 insertarABB x Nil = Bin Nil x Nil
3 insertarABB x (Bin i r d) = if x < r
4     then Bin (insertarABB x i) r d
5     else Bin i r (insertarABB x d)
```

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

Tarea: prueben escribir estas funciones con los esquemas de recursión correspondientes.

Folds sobre otras estructuras

Dado el siguiente tipo que representa polinomios:

```
1 data Polinomio a = X
2                  | Cte a
3                  | Suma (Polinomio a) (Polinomio a)
4                  | Prod (Polinomio a) (Polinomio a)
```

- Definir la función `evaluar :: Num a => a -> Polinomio a -> a`

Folds sobre otras estructuras

Dado el siguiente tipo que representa polinomios:

```
1 data Polinomio a = X
2                  | Cte a
3                  | Suma (Polinomio a) (Polinomio a)
4                  | Prod (Polinomio a) (Polinomio a)
```

- Definir la función `evaluar :: Num a => a -> Polinomio a -> a`
- Definir el esquema de recursión estructural `foldPoli` para polinomios (y dar su tipo).

Folds sobre otras estructuras

Dado el siguiente tipo que representa polinomios:

```
1 data Polinomio a = X
2                  | Cte a
3                  | Suma (Polinomio a) (Polinomio a)
4                  | Prod (Polinomio a) (Polinomio a)
```

- Definir la función `evaluar :: Num a => a -> Polinomio a -> a`
- Definir el esquema de recursión estructural `foldPoli` para polinomios (y dar su tipo).
- Redefinir `evaluar` usando `foldPoli`.

Una estructura más compleja

Dado el tipo de datos

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

de árboles donde cada nodo tiene una cantidad indeterminada de hijos.

1. Escribir el esquema de recursión estructural para `RoseTree`.
2. Usando el esquema definido, escribir las siguientes funciones:
   - `hojas`, que dado un `RoseTree`, devuelva una lista con sus hojas ordenadas de izquierda a derecha, según su aparición en el `RoseTree`.
   - `ramas`, que dado un `RoseTree`, devuelva los caminos de su raíz a cada una de sus hojas.
   - `tamaño`, que devuelve la cantidad de nodos de un `RoseTree`.
   - `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.

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 pertenece a c y `False` en caso contrario.

1. Definir la constante `vacío :: Conj a`, y la función `agregar :: Eq a => a -> Conj a -> Conj a`.

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 pertenece a c y `False` en caso contrario.

1. Definir la constante `vacío :: Conj a`, y la función `agregar :: Eq a => a -> Conj a -> Conj a`.
2. Escribir las funciones `intersección`, `unión` y `diferencia` (todas de tipo `Conj a -> Conj a-> Conj a`).

Fin

¿¿¿¿¿¿¿¿¿?????????