Esquemas de recursión sobre listas

Programación funcional

Repaso

- Reducción.
- Normalización, confluencia y transparencia referencial.
- Currificación y funciones de alto orden.
- Tipos algebráicos, recursión e inducción.

Map

```
incl :: [Int] -> [Int]
incl [] = []
incl (n:ns) = (1+) n : incl ns

upl :: [Char] -> [Char]
upl [] = []
upl (c:cs) = upper c : upl cs

nulls :: [[a]] -> [Bool]
nulls [] = []
nulls (xs:xss) = null xs : nulls xss
```

Map

```
incl :: [Int] -> [Int]
incl [] = []
incl (n:ns) = (1+) n : incl ns

upl :: [Char] -> [Char]
upl [] = []
upl (c:cs) = upper c : upl cs

nulls :: [[a]] -> [Bool]
nulls [] = []
nulls (xs:xss) = null xs : nulls xss

map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs

incl'  = map (1+)
upl'   = map upper
nulls' = map null
```

¿Son equivalentes?

```
incl ≡ incl'
```

Por PE tiene que valer para todo `xs::[Int]`:

```
incl xs ≡ incl' xs
incl' ⇒ incl xs ≡ map (1+) xs
```

Por inducción sobre `xs::[Int]`:

- Caso Base:

```
incl [] ≡ map (1+) []
incl.1, map.1 ⇒ [] ≡ []
```

- Caso Inductivo:

```
incl (x:xs) ≡ map (1+) (x:xs)
incl.2, map.2 ⇒ (1+) x : incl xs ≡ (1+) x : map (1+) xs
HI ⇒ (1+) x : incl xs ≡ (1+) x : incl xs
```

Filter

```
gt0 :: [Int] -> [Int]
gt0 [] = []
gt0 (n:ns) = (if (>0) n then [n] else []) ++ gt0 ns

digits :: [Char] -> [Char]
digits [] = []
digits (c:cs) = (if isDigit c then [c] else []) ++ digits cs

nnulls :: [[a]] -> [[a]]
nnulls [] = []
nnulls (xs:xss) = (if (not . null) xs then [xs] else []) ++
                  nnulls xs
```

Filter

```
gt0 :: [Int] -> [Int]
gt0 [] = []
gt0 (n:ns) = (if (>0) n then [n] else []) ++ gt0 ns

digits :: [Char] -> [Char]
digits [] = []
digits (c:cs) = (if isDigit c then [c] else []) ++ digits cs

nnulls :: [[a]] -> [[a]]
nnulls [] = []
nnulls (xs:xss) = (if (not . null) xs then [xs] else []) ++
                  nnulls xs

filter :: (a -> Bool) -> [a] -> [a]
filter f [] = []
filter f (x:xs) = (if f x then [x] else []) ++ filter f xs

gt0'    = filter (>0)
digits' = filter isDigit
nnulls' = filter (not . null)
```

Fold

```
sum :: [Int] -> Int
sum [] = 0
sum (n:ns) = (+) n (sum ns)

all :: [Bool] -> Bool
all [] = True
all (b:bs) = (&&) b (all bs)

concat :: [[a]] -> [a]
concat [] = []
concat (xs:xss) = (++) xs (concat xss)
```

Fold

```
sum :: [Int] -> Int
sum [] = 0
sum (n:ns) = (+) n (sum ns)

all :: [Bool] -> Bool
all [] = True
all (b:bs) = (&&) b (all bs)

concat :: [[a]] -> [a]
concat [] = []
concat (xs:xss) = (++) xs (concat xss)

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

sum'    = foldr (+) 0
all'    = foldr (&&) True
concat' = foldr (++) []
```

Propiedades de esquemas

Fusión

Si `h (f x y)` $\equiv$ `g x (h y)`, entonces:

`h . foldr f z` $\equiv$ `foldr g (h z)`

Propiedades de esquemas

Fusión

Si `h (f x y)` $\equiv$ `g x (h y)`, entonces:

`h . foldr f z` $\equiv$ `foldr g (h z)`

Ejemplo:

```
(+1) . sum = foldr (+) 1
```

Propiedades de esquemas

Fusión

Si `h (f x y)` ≡ `g x (h y)`, entonces:

`h . foldr f z` ≡ `foldr g (h z)`

Ejemplo:

```
(+1) . sum = foldr (+) 1
```

Demostración, por inducción en `xs::[a]`:

```
h (foldr f z xs)  ≡  foldr g (h z) xs
```

- Caso Base:

```
h (foldr f z [])  ≡  foldr g (h z) []
foldr.1 => h z  ≡  h z
```

- Caso Inductivo:

```
h (foldr f z (x:xs))  ≡  foldr g (h z) (x:xs)
foldr.2 => h (f x (foldr f z xs))  ≡  g x (foldr g (h z) xs)
HI => h (f x (foldr f z xs))  ≡  g x (foldr f z xs)
Verdadero ya que h (f x y)  ≡  g x (h y)
```

Optimizaciones

```
(map f . map g) ≡ map (f . g)

filter f . filter g ≡ filter (\x -> f x && g x)

foldr f z . map g ≡ foldr (f . g) z

map f . concat ≡ concatMap f
```

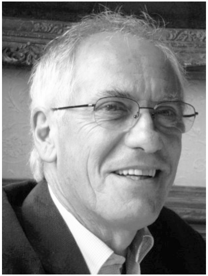
Listas por comprensión

```
map f xs ≡ [ f x | x <- xs ]

filter f xs ≡ [ x | x <- xs, f x ]

concatMap f xss ≡ [ f x | xs <- xss, x <- xs ]
```

Variantes de fold

- Asociativa a izquierda
- Recursiva a la cola

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



Richard Bird

Dualidad:

Si `f :: (a -> b -> b)` asociativa, entonces para toda lista finita:

`foldr f z` ≡ `foldl f z`

Desafío

Reescribir usando `foldr` (y sin auxiliares):

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f [x:xs] [y:ys] = f x y : zipWith f xs ys
zipWith f _ _ = []
```