

# Programación Funcional Avanzada

## Morfismos

Ernesto Hernández-Novich  
<emhn@usb.ve>

Universidad “Simón Bolívar”

Copyright ©2010-2016

# Morfismos

## Patrones de recursividad

- Catamorfismos – Colapsan una estructura a un valor (*fold*)
  - Paramorfismos – usando recursión primitiva.
  - Zygomorfismos – con una función auxiliar.
  - Histomorfismos – aprovechando cálculos previos.
  - ...



# Morfismos

## Patrones de recursividad

- Catamorfismos – Colapsan una estructura a un valor (*fold*)
  - Paramorfismos – usando recursión primitiva.
  - Zygomorfismos – con una función auxiliar.
  - Histomorfismos – aprovechando cálculos previos.
  - ...
- Anamorfismos – Generan una estructura a partir de un valor (*unfold*)
  - Apomorfismos – un nivel por paso.
  - Futumorfismos – varios niveles por paso.
  - Postpromorfismos – usando transformaciones naturales.
  - ...



# Morfismos

## Patrones de recursividad

- Catamorfismos – Colapsan una estructura a un valor (*fold*)
  - Paramorfismos – usando recursión primitiva.
  - Zygomorfismos – con una función auxiliar.
  - Histomorfismos – aprovechando cálculos previos.
  - ...
- Anamorfismos – Generan una estructura a partir de un valor (*unfold*)
  - Apomorfismos – un nivel por paso.
  - Futumorfismos – varios niveles por paso.
  - Postpromorfismos – usando transformaciones naturales.
  - ...
- Hylomorfismos – anamorfismo seguido de catamorfismo (*refold*)
  - Cronomorfismo – futumorfismo seguido de histomorfismo.
  - Sincromorfismo – usando una estructura intermedia.
  - ...

# fold

## Sobre listas

- “Por izquierda” o “por derecha”.

```
foldr step b []          = b
foldr step b (x:xs)     = step x $ foldr step b xs

foldl step b []          = b
foldl step b (x:xs)     = foldl step (step b x) xs
```



# fold

## Sobre listas

- “Por izquierda” o “por derecha”.

```
foldr step b []          = b
foldr step b (x:xs)     = step x $ foldr step b xs

foldl step b []          = b
foldl step b (x:xs)     = foldl step (step b x) xs
```

- ¡Cuidado con la flojera del lenguaje!
  - La evaluación genera *thunks* – puede conducir a *space leaks*.
  - `Data.List` provee `foldl'` que es estricto al evaluar `step` – es “más estricto” pero sólo por una evaluación.



# fold

## Sobre listas

- “Por izquierda” o “por derecha”.

```
foldr step b []          = b
foldr step b (x:xs)     = step x $ foldr step b xs

foldl step b []          = b
foldl step b (x:xs)     = foldl step (step b x) xs
```

- ¡Cuidado con la flojera del lenguaje!
  - La evaluación genera *thunks* – puede conducir a *space leaks*.
  - `Data.List` provee `foldl'` que es estricto al evaluar `step` – es “más estricto” pero sólo por una evaluación.
- El valor resultado puede ser estructurado.

# fold

## Sobre listas

- “Por izquierda” o “por derecha”.

```
foldr step b []          = b
foldr step b (x:xs)     = step x $ foldr step b xs

foldl step b []          = b
foldl step b (x:xs)     = foldl step (step b x) xs
```

- ¡Cuidado con la flojera del lenguaje!
  - La evaluación genera *thunks* – puede conducir a *space leaks*.
  - `Data.List` provee `foldl'` que es estricto al evaluar `step` – es “más estricto” pero sólo por una evaluación.
- El valor resultado puede ser estructurado.
- Uno de ellos sólo puede operar con listas finitas – ¿cuál?





# fold – Ejemplos

## Hacia valores simples

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

```
prod :: Num a => [a] -> a
prod = foldr (*) 1
```

```
all :: [a] -> Bool
all = foldl and True
```

```
any :: [a] -> Bool
any = foldl or False
```



# fold – Ejemplos

## Hacia valores complejos

```
map :: (a -> b) -> [a] -> [b]
map f = foldr ((:).f) []
```

```
(++) :: [a] -> [a] -> [a]
xs ++ ys = foldr (:) ys xs
```

```
concat :: [[a]] -> [a]
concat = foldr (++) []      -- Buena
concat = foldl (++) []     -- Mala
```

```
compose :: [a -> a] -> (a -> a)
compose = foldr (.) id
```



# fold – Ejemplos

En la vida real

## Insertion Sort

- Tomar el primer elemento y colocarlo “en su lugar”.
- Repetir para todos los elementos de la lista.



# fold – Ejemplos

En la vida real

## Insertion Sort

- Tomar el primer elemento y colocarlo “en su lugar”.
- Repetir para todos los elementos de la lista.

```

isort :: Ord a => [a] -> [a]
isort = foldr ins []
  where ins x []      = [x]
        ins x (y:ys) = if x < y then (x:y:ys)
                        else y : ins x ys
  
```



# fold – Ejemplos

## En la vida real

- Se recorre un archivo para construir [(String,Datos)].
- Es necesario consultar muchas veces usando el primer elemento como clave para encontrar el valor asociado.
  - Usar lookup es prohibitivo por ser  $O(n)$ .
  - Data.Map provee una tabla de hash con  $O(\log n)$ .
- En Data.Map encontramos – (key-value)
  - `empty :: Map k v`
  - `insert :: k -> v -> Map k v -> Map k v`



# fold – Ejemplos

## En la vida real

- Se recorre un archivo para construir [(String,Datos)].
- Es necesario consultar muchas veces usando el primer elemento como clave para encontrar el valor asociado.
  - Usar lookup es prohibitivo por ser  $O(n)$ .
  - Data.Map provee una tabla de hash con  $O(\log n)$ .
- En Data.Map encontramos – (key-value)
  - `empty :: Map k v`
  - `insert :: k -> v -> Map k v -> Map k v`

```
import qualified Data.Map as Map

loadAll datos = foldl' carga Map.empty datos
  where carga m d = Map.insert (fst d) (snd d) m
```



# fold – Ejemplos

## En la vida real

Calcular los estadísticos básicos de una lista de números:

- Máximo y mínimo.
- Promedio.
- Varianza y desviación estándar.

¿Podemos hacerlo en **una** pasada?

# fold – Ejemplos

## En la vida real

Calcular los estadísticos básicos de una lista de números:

- Máximo y mínimo.
- Promedio.
- Varianza y desviación estándar.

¿Podemos hacerlo en **una** pasada?

```
stats (x:xs) = fin . foldl' go (x,x,x,x*x,1) $ xs
  where go (mx,          mn,          s,          ss,          n ) x =
          (max x mx, min x mx, s+x, ss+x*x, n+1)
  fin (mx,mn,s,ss,n) = (mx,mn,av,va,stdev,n)
    where av      = s/n
          va      = (1/(n-1))*ss - (n/(n-1))*av*av
          stdev   = sqrt va
```



# fold genérico

... existe para cualquier tipo de datos

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```



# fold genérico

... existe para cualquier tipo de datos

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

- Una función por cada constructor del tipo ...

```
Leaf    :: a -> Tree a
```

```
Branch :: Tree a -> Tree a -> Tree a
```



# fold genérico

... existe para cualquier tipo de datos

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

- Una función por cada constructor del tipo ...

```
Leaf    :: a -> Tree a
```

```
Branch :: Tree a -> Tree a -> Tree a
```

- ... que debe colapsar hacia el tipo resultado

```
fleaf   :: a -> b
```

```
fbranch :: b -> b -> b
```



# fold genérico

... existe para cualquier tipo de datos

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

- Una función por cada constructor del tipo ...

```
Leaf    :: a -> Tree a
Branch  :: Tree a -> Tree a -> Tree a
```

- ... que debe colapsar hacia el tipo resultado

```
fleaf   :: a -> b
fbranch :: b -> b -> b
```

- Entonces el fold genérico para Tree a queda

```
foldT :: (b -> b -> b) -> (a -> b) -> Tree a -> b
foldT fbranch fleaf = go
  where go (Leaf x)      = fleaf x
        go (Branch l r) = fbranch (go l) (go r)
```

# fold genérico

Y luego podemos usarlo convenientemente

- Sumar los valores almacenados en el árbol

```
sumTree = foldT (+) id
```

- De cada hoja sólo interesa su valor.
- De cada bifurcación, obtenemos la suma.



# fold genérico

Y luego podemos usarlo convenientemente

- Sumar los valores almacenados en el árbol

```
sumTree = foldT (+) id
```

- De cada hoja sólo interesa su valor.
  - De cada bifurcación, obtenemos la suma.
- Obtener los valores de las hojas de izquierda a derecha

```
fringeTree = foldT (++) (\x -> [x])
```

- Cada hoja debemos “meterla” en una lista
  - Cada bifurcación debe concatenar las listas hijas.



# fold absurdamente pedagógico

```
data WTF a b = Foo a
              | Bar b
              | Baz [(a,b)]
              | Qux (WTF a b)
```



# fold absurdamente pedagógico

```
data WTF a b = Foo a
             | Bar b
             | Baz [(a,b)]
             | Qux (WTF a b)
```

- Notamos los tipos de cada constructor...

```
Foo  :: a -> WTF a b
Bar  :: b -> WTF a b
Baz  :: [(a,b)] -> WTF a b
Qux  :: WTF a b -> WTF a b
```



# fold absurdamente pedagógico

```
data WTF a b = Foo a
             | Bar b
             | Baz [(a,b)]
             | Qux (WTF a b)
```

- Notamos los tipos de cada constructor...

```
Foo  :: a -> WTF a b
Bar  :: b -> WTF a b
Baz  :: [(a,b)] -> WTF a b
Qux  :: WTF a b -> WTF a b
```

- ...que deben colapsar a un nuevo tipo resultado c

```
foo  :: a -> c
bar  :: b -> c
baz  :: [(a,b)] -> c
qux  :: c -> c
```

# fold absurdamente pedagógico

```

wtFold :: (a -> c) ->
         (b -> c) ->
         ([[a,b]] -> c)
         -> (c -> c)
         -> WTF a b
         -> c

wtFold foo bar baz qux = go
  where go (Foo x)      = foo x
        go (Bar x)     = bar x
        go (Baz xs)    = baz xs
        go (Qux wtf)   = qux $ wtFold foo bar baz qux wtf

```



# unfold

## Generador de listas a partir de una semilla

- Una sola posibilidad

```
unfoldr test b =  
  case test b of  
    Just (v,b') -> v : unfoldr test b'  
    Nothing      -> []
```

- La función test decide
  - Nothing more – terminar la lista.
  - Just add this and carry on – continuar generando.
- ¡La lista puede ser infinita!
- La semilla puede ser un valor estructurado.



# unfold – Ejemplos

```
fibs :: [Int]
fibs = unfoldr (\(f2,f1) -> Just (f2+f1,(f1,f2+f1)))
              (0,1)

iterate :: (a -> a) -> a -> [a]
iterate f = unfoldr (\x -> Just (x, f x))
```

# unfold – Ejemplos

En la vida real

## Selection Sort

- Buscar el elemento más pequeño y colocarlo de primero.
- Repetir para todos los elementos de la lista.



# unfold – Ejemplos

En la vida real

## Selection Sort

- Buscar el elemento más pequeño y colocarlo de primero.
- Repetir para todos los elementos de la lista.

```
ssort :: Ord a => [a] -> [a]
ssort = unfoldr delmin
  where delmin [] = Nothing
        delmin xs = Just (y, delete y xs)
          where y = minimum xs
```



# unfold genérico

... existe para cualquier tipo de datos

- La función de control es más compleja
  - Terminación usa *alguna* de los constructores no-recursivos.
  - Se necesita una semilla para cada caso recursivo.



# unfold genérico

... existe para cualquier tipo de datos

- La función de control es más compleja
  - Terminación usa *alguna* de los constructores no-recursivos.
  - Se necesita una semilla para cada caso recursivo.
- En el caso simple – que es el de las listas. . .
  - Un constructor no-recursivo con aridad 0.
  - Un constructor recursivo con aridad N.
  - Se usa  $b \rightarrow \text{Maybe } s$ , donde  $s$  es el tipo de la semilla.





# unfold genérico

... existe para cualquier tipo de datos

- La función de control es más compleja
  - Terminación usa *alguno* de los constructores no-recursivos.
  - Se necesita una semilla para cada caso recursivo.
- En el caso simple – que es el de las listas. . .
  - Un constructor no-recursivo con aridad 0.
  - Un constructor recursivo con aridad N.
  - Se usa  $b \rightarrow \text{Maybe } s$ , donde  $s$  es el tipo de la semilla.
- Caso complejo
  - Constructores no-recursivos con aridades varias.
  - Constructores recursivos, posiblemente usando valores simples.
  - Se necesita un tipo algebraico para casos base y semillas.



# unfold genérico

... existe para cualquier tipo de datos

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

- Leaf – constructor no-recursivo unario.
- Branch – constructor recursivo que requiere dos semillas.
- Aprovecho el tipo estándar Either a (b,b)
  - Left envolverá los valores finales.
  - Right envolverá las semillas.



# unfold genérico

... existe para cualquier tipo de datos

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

- Leaf – constructor no-recursivo unario.
- Branch – constructor recursivo que requiere dos semillas.
- Aprovecho el tipo estándar Either a (b,b)
  - Left envolverá los valores finales.
  - Right envolverá las semillas.

```
unfoldT :: (b -> Either a (b,b)) -> b -> Tree a
unfoldT test b =
  case test b of
    Left x          -> Leaf x
    Right (b1,b2)  -> Branch (unfoldT test b1)
                          (unfoldT test b2)
```

test es la función provista por el usuario



# unfold absurdamente pedagógico

```
data WTF a b = Foo a
              | Bar b
              | Baz [(a,b)]
              | Qux (WTF a b)
```

... se viene un tipo especial



# unfold absurdamente pedagógico

```
data WTF a b = Foo a
             | Bar b
             | Baz [(a,b)]
             | Qux (WTF a b)
```

... se viene un tipo especial

```
data WTH a b c = Afoo a
               | Abar b
               | Abaz [(a,b)]
               | Aqux c
```

```
unWTFold :: (c -> WTH a b c) -> c -> WTF a b
unWTFold test s =
  case test s of
    Afoo x   -> Foo x
    Abar x   -> Bar x
    Abaz xs  -> Baz xs
    Aqux s'  -> Qux (unWTFold test s')
```

# The fast Monoid

## Usefulness unfolds

- `Data.Sequence` provee la construcción iterativa

```
unfoldr :: (b -> Maybe (a,b)) -> b -> Seq a
unfoldl :: (b -> Maybe (a,b)) -> b -> Seq a
```

- `unfoldr` no termina si se intenta generar una secuencia infinita.
  - La generación es  $O(n)$  en ambos sentidos.
- Usualmente es *más* eficiente usar estos, que construir una lista y convertirla en `Seq a`.



# refold

- Una secuencia de expansiones y colapsos.
  - Comenzar con `unfold` para generar una estructura de datos.
  - Continuar con un `fold` para colapsarla.
  - ... tantas veces como haga falta.
- Corresponde al paradigma **productor-consumidor**.
- Lo hace un compilador (fuente  $\rightarrow$  AST  $\rightarrow$  TAC  $\rightarrow$  objeto).
- En ocasiones puede evitarse el paso intermedio – (*deforestación*).
  - **Teoremas de Fusión** para la combinación de `map`, `fold` y `unfold` pueden aplicarse para reducir o eliminar la estructura intermedia.
  - Incorporar esos teoremas en compiladores de lenguajes funcionales es un tópico de investigación actual.



# refold – Ejemplos

```
fac =
  foldl' (*) 1 .
  unfoldr (\x -> if x == 0
                 then Nothing
                 else Just (x,pred x))

tobin =
  foldl' (flip (:)) [] .
  unfoldr (\x -> if x == 0
                 then Nothing
                 else Just (x 'mod' 2, x 'div' 2))
```





# Foldable – colapso generalizado

*Typeclass* provisto en `Data.Foldable`

```
class Foldable t where
  fold      :: (Monoid m) => t m -> m
  foldMap  :: (Monoid m) => (a -> m) -> t a -> m
  foldr    :: (a -> b -> b) -> b -> t a -> b
  foldl    :: (a -> b -> a) -> a -> t b -> a
  foldr1   :: (a -> a -> a) -> t a -> a
  foldl1   :: (a -> a -> a) -> t a -> a
```

- Generalizando a `Monoid` – resultados en `[a]`, `Seq a`, `Sum a...`
- `Foldable` abstrae contenedores “colapsables” a un valor – operaciones generales independientes del contenedor.
- Sólo es necesario programar `foldMap` o `foldr` – el compilador *genera* el resto automáticamente.



# ¿Cómo me ayuda usar Foldable?

## Origami en drogas

- Funciones `fold*` de `Prelude` o `Data.List` sólo procesan listas – `fold*` de `Data.Foldable` procesa *cualquier* tipo de datos instanciado en esa clase.



# ¿Cómo me ayuda usar Foldable?

## Origami en drogas

- Funciones `fold*` de `Prelude` o `Data.List` sólo procesan listas – `fold*` de `Data.Foldable` procesa *cualquier* tipo de datos instanciado en esa clase.
- ¡Santos colapsos generalizados!

```

all      :: (Foldable t) => (a -> Bool) -> t a -> Bool
any      :: (Foldable t) => (a -> Bool) -> t a -> Bool
find     :: (Foldable t) => (a -> Bool) -> t a -> Maybe a
and      :: (Foldable t) => t Bool -> Bool
concatMap :: (Foldable t) => (a -> [b]) -> t a -> [b]
elem     :: (Foldable t, Eq a) => a -> t a -> Bool
maximum  :: (Foldable t, Ord a) => t a -> a
sum      :: (Foldable t, Num a) => t a -> a
...
sequence_ :: (Foldable t, Monad m) => t (m a) -> m ()
  
```

# ¿Qué cosas son “colapsables”?

- `Data.Foldable` define instancias para
  - `List` – doh!
  - `Maybe`
  - `Array`
- `Data.Map` define instancia de `Foldable`
- `Data.Set` define instancia de `Foldable`
- `Data.Tree` define instancia de `Foldable`
- `Data.Sequence` define instancia de `Foldable` – alternativa a `Data.List` simétrica y eficiente.
- Importar `Data.Foldable` calificado – usar `DF.any`, `DF.sum`, ...



# ¿Qué cosas son “colapsables”?

- `Data.Foldable` define instancias para
  - `List` – doh!
  - `Maybe`
  - `Array`
- `Data.Map` define instancia de `Foldable`
- `Data.Set` define instancia de `Foldable`
- `Data.Tree` define instancia de `Foldable`
- `Data.Sequence` define instancia de `Foldable` – alternativa a `Data.List` simétrica y eficiente.
- Importar `Data.Foldable` calificado – usar `DF.any`, `DF.sum`, ...

Si tu tipo de datos requiere ser colapsado, instanciar `Foldable` te ahorra *mucho* trabajo.



# Meanwhile in GHC

... a partir de 6.12

```
{-# LANGUAGE DeriveFunctor, DeriveFoldable #-}  
  
import qualified Data.Foldable as DF  
  
data WaitWhat a = OMG a  
                | Pones [WaitWhat a]  
                deriving (Show, DF.Foldable)  
  
wtf = Pones [OMG 1, Pones [OMG 2, OMG 3], OMG 4]
```

- GHC es capaz de *generar* la instancia `Foldable` – suficiente en el caso general y sin esfuerzo de programación.
- No se puede generar si la instancia de `Functor` fue escrita “a mano”.

# Meanwhile in GHC

... a partir de 6.12

```
{-# LANGUAGE DeriveFunctor, DeriveFoldable #-}

import qualified Data.Foldable as DF

data WaitWhat a = OMG a
                | Ponies [WaitWhat a]
                deriving (Show, DF.Foldable)

wtf = Ponies [OMG 1, Ponies [OMG 2, OMG 3], OMG 4]
```

- GHC es capaz de *generar* la instancia `Foldable` – suficiente en el caso general y sin esfuerzo de programación.
- No se puede generar si la instancia de `Functor` fue escrita “a mano”.

# Shut up and take my money!



# Quiero saber más...

- [Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire](#)  
Meijer, Fokkinga & Paterson
- [Catamorfismo – Wikipedia](#)
- [Anamorfismo – Wikipedia](#)
- [Hylomorfismo – Wikipedia](#)
- [Documentación de Data.Foldable](#)

