

Hylomorfismos

Agustín Fernández Bergé

Diciembre 2025

En programación funcional es común usar estructuras de datos que se definen de manera inductiva:

```
data List a = Nil | Cons a (List a)
```

```
data Tree a = E | L a | N (Tree a) a (Tree a)
```

```
data MathExpr = Num Real | Add MathExpr MathExpr | Mul MathExpr MathExpr
```

Esto implica que podemos escribir *algoritmos recursivos* sobre estas estructuras:

```
eval :: MathExpr -> Real
```

```
eval (Real r) = r
```

```
eval (Add a b) = eval a + eval b
```

```
eval (Mul a b) = eval a * eval b
```

```
-- Uso la notación alternativa de listas, List a = [] | (a: List a)
```

```
inorder :: Tree a -> List a
```

```
inorder E = []
```

```
inorder (L x) = [x]
```

```
inorder (N l x r) = inorder l ++ [x] ++ inorder r
```

```
qsort :: Ord a => [a] -> [a]
```

```
qsort [] = []
```

```
qsort (p:xs) = (qsort smaller) ++ [p] ++ (qsort larger)
```

```
  where
```

```
    smaller = [x | x <- xs, x < p]
```

```
    larger  = [x | x <- xs, x >= p]
```

Muchos de estos algoritmos son similares entre sí. Por ejemplo están aquellos que consumen una estructura para generar un valor:

```
sum :: List Real -> Real
sum [] = 0
sum (x:xs) = x + sum xs
```

```
mul :: List Real -> Real
mul [] = 1
mul (x:xs) = x * mul xs
```

Podemos implementar ambos algoritmos bajo un mismo patrón denominado *foldr*:

```
foldr :: (a->b->b) -> b -> List a -> b  
foldr op e [] = e  
foldr op e (x:xs) = op x (foldr op e xs)
```

sum y mul son simplemente:

```
sum xs = foldr (+) 0 xs  
mul xs = foldr (*) 1 xs
```

Esto nos permite:

- Reciclar código
- Mejorar la legibilidad
- Facilitar pruebas de corrección
- Aplicar optimizaciones

No todos los algoritmos siguen este patrón. Un ejemplo es `makeTree`, que construye una estructura a partir de un valor.

```
makeTree :: Int -> Int -> Tree Int
```

```
makeTree a b
```

```
  | a>=b = E
```

```
  | a+1==b = L a
```

```
  | otherwise = let
```

```
      m = (a+b) `div` 2
```

```
  in
```

```
      N (makeTree a m) m (makeTree (m+1) b)
```

Sin embargo, muchos de los algoritmos de recursión sobre estructuras se pueden unificar bajo un mismo patrón, los **hylomorfismos**.

Definición

Dada una categoría \mathcal{C} y un endofunctor $F : \mathcal{C} \rightarrow \mathcal{C}$ denominado *functor base*, una **F-álgebra** es un par (A, α) donde:

- A es un objeto de \mathcal{C} denominado *carrier* (portador)
- $\alpha : FA \rightarrow A$ es un morfismo en \mathcal{C} denominado *acción* o *evaluator*

Un **morfismo** entre dos F-álgebras (A, α) y (B, β) es un morfismo $f : A \rightarrow B$ en \mathcal{C} tal que el siguiente diagrama conmuta:

$$\begin{array}{ccc} FA & \xrightarrow{Ff} & FB \\ \alpha \downarrow & & \downarrow \beta \\ A & \xrightarrow{f} & B \end{array}$$

Las F-álgebras y sus morfismos forman una categoría denominada *categoría de F-álgebras* y denotada como $F\text{-}\mathbf{Alg}(\mathcal{C})$.

De manera dual, una **F-coálgebra** en una categoría \mathcal{C} es un par (A, α) donde:

- A es un objeto de \mathcal{C} denominado *carrier*
- $\alpha : A \rightarrow FA$ es un morfismo en \mathcal{C} denominado *acción* o *generador*

Un morfismo entre dos F-coálgebras (A, α) y (B, β) es un morfismo $f : A \rightarrow B$ en \mathcal{C} tal que el siguiente diagrama conmuta:

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ \alpha \downarrow & & \downarrow \beta \\ FA & \xrightarrow{Ff} & FB \end{array}$$

La categoría de F-coálgebras se denota como $F\text{-Coalg}(\mathcal{C})$.

Álgebras iniciales y cólgebras terminales

Al objeto inicial de la categoría de F -álgebras, si existe, se le denomina **álgebra inicial** y se denota como $(\mu F, \text{in}_F)$.

Al único morfismo de F -álgebras de $(\mu F, \text{in}_F)$ a cualquier otra F -álgebra (A, α) se le denomina **catamorfismo** o simplemente **fold** y se denota como $(|\alpha|) : \mu F \rightarrow A$ (usando *banana brackets*).

De manera dual, al objeto terminal de la categoría de F -cólgebras, si existe, se le denomina **cólgebra terminal** y se denota como $(\nu F, \text{out}_F)$.

Al único morfismo de F -cólgebras de cualquier otra F -cólgebra (A, α) a $(\nu F, \text{out}_F)$ se le denomina **anamorfismo** o **unfold** y se denota como $|\alpha| : A \rightarrow \nu F$ (usando *lens brackets*).

Estructuras de datos como F-álgebras y F-coálgebras

Muchas estructuras de datos pueden definirse como un álgebra o una coálgebra donde el *carrier* es un punto fijo del funtor base.

Punto fijo de un endofuntor

Un objeto X en una categoría \mathcal{C} es un **punto fijo** de un endofuntor $F : \mathcal{C} \rightarrow \mathcal{C}$ si existe un isomorfismo $X \cong FX$.

El carrier de toda álgebra inicial o coálgebra terminal es un punto fijo de su funtor base. En este caso, los catamorfismos(respectivamente anamorfismos) inducen un esquema de recursión que permiten consumir(respectivamente construir) dichas estructuras de datos.

Ejemplo: Listas de tipo a

Podemos ver a los tipos del lenguaje Haskell como objetos de una categoría denominada **Hask**, donde los morfismos son las funciones totales entre dichos tipos.

Un ejemplo de funtor en **Hask** es el siguiente:

```
data ListF a x = NilF | ConsF a x
  deriving Functor
```

Con esta definición, dado un tipo a fijo, podemos crear nuevos tipos de datos:

- `ListF a Int`
- `ListF a Real`
- `ListF a (Tree a)`
- etc...

Además el lenguaje infiere automáticamente la definición del funtor sobre morfismos:

```
fmap :: (x -> y) -> (ListF a x -> ListF a y)
fmap f = ff
  where
    ff NilF          = NilF
    ff (ConsF a x)   = ConsF a (f x)
```

Dicha función cumple las leyes de los funtores:

- `fmap id = id`
- `fmap (g . f) = fmap g . fmap f`

La definición inductiva de listas de tipo `a` que vimos antes:

```
data List a = Nil | Cons a (List a)
```

Es un punto fijo del endofunctor `ListF a`:

```
ListF a (List a) = NilF | ConsF a (List a) === List a
```

Más aún, `List a` junto con la función:

```
inn :: ListF a (List a) -> List a
inn NilF          = Nil
inn (ConsF a xs) = Cons a xs
```

constituyen el álgebra inicial de `ListF a`.

Por lo que se puede definir el catamorfismo desde `List a` hacia cualquier otra álgebra de acción `a`:

```
cata :: (ListF a b -> b) -> (List a -> b)
cata a = a . fmap (cata a) . innInv
```

A grandes rasgos, dado un endofunctor F sobre una categoría \mathcal{C} tenemos que:

- El endofunctor F puede dar una cierta “estructura recursiva” a los objetos de \mathcal{C}
- El mismo endofunctor F aplicado a un morfismo de \mathcal{C} nos da un esquema de recursión sobre dicha estructura
- Las F -álgebras nos dan una forma de **consumir** dicha estructura recursiva para obtener un objeto *carrier*
- Las F -coálgebras permiten definir funciones que **generan** dicha estructura recursiva a partir de un objeto *carrier*

La mayoría de los esquemas de recursión estructurada siguen una temática “Divide & Conquer”:

1. Se descompone un problema en subproblemas más pequeños (Divide)
2. Se resuelve recursivamente cada uno de los subproblemas
3. Se combinan las soluciones de los subproblemas para obtener la solución del problema original (Conquer)

Podemos usar lo visto hasta ahora para definir este esquema de recursión de manera generalizada como un **hylomorfismo**.

Definición

Sea F un endofunctor sobre una categoría \mathcal{C} , (A, α) una F -álgebra y (C, γ) una F -coálgebra. Un morfismo $h : C \rightarrow A$ en \mathcal{C} es un **hylomorfismo** (o un homomorfismo de coálgebra a álgebra) si satisface la siguiente **hyloecuación**:

$$h = \alpha \circ F h \circ \gamma$$

Es decir, un hylomorfismo hace conmutar el siguiente diagrama:

$$\begin{array}{ccc} C & \xrightarrow{h} & A \\ \gamma \downarrow & & \uparrow \alpha \\ FC & \xrightarrow{Fh} & FA \end{array}$$