Hylomorfismos

Agustín Fernández Bergé

19 de Diciembre 2025

Motivación

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:

```haskell
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 $\mathscr{C}$ y un endofuntor $F : \mathscr{C} \to \mathscr{C}$ denominado *funtor base*, una **F-álgebra** es un par (A, α) donde:

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

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

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

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

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

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

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

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

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

Álgebras iniciales y coálgebras terminales

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

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

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

Al único morfismo de F-coálgebras de cualquier otra F-coálgebra (A, α) a $(\nu F, \mathsf{out}_F)$ se le denomina **anamorfismo** o **unfold** y se denota como $|(\alpha)| : A \to \nu F$ (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 $\mathscr{C}$ es un **punto fijo** de un endofuntor $F : \mathscr{C} \to \mathscr{C}$ 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 endofuntor `ListF a`:

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

Mas 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 endofuntor F sobre una categoría $\mathscr{C}$ tenemos que:

- El endofuntor F puede dar una cierta "estructura recursiva" a los objetos de $\mathscr{C}$
- El mismo endofuntor F aplicado a un morfismo de $\mathscr{C}$ 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**.

Hylomorfismos

Definición
Sea F un endofuntor sobre una categoría $\mathscr{C}$, (A, α) una F-álgebra y (C, γ) una F-coálgebra. Un morfismo $h : C \to A$ en $\mathscr{C}$ 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:

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

El conjunto de todos los hylomorfismos entre la F-coálgebra (C, γ) y la F-álgebra (A, α) se denota como $\mathrm{Hylo}((C, \gamma), (A, \alpha))$ o simplemente $\mathrm{Hylo}_{\gamma}^{\alpha}$.

Ejemplo: Quicksort

El algoritmo de ordenamiento rápido (quicksort) se puede definir como un hylomorfismo que captura perfectamente el paradigma "Divide & Conquer":

Consideremos el endofuntor QsortF definido como:

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

fmap :: (x->y) -> QsortF a x -> QsortF a y
fmap f = ff
  where
    ff NilF           = NilF
    ff (ConsF l p r)  = ConsF (f l) p (f r)
```

La acción de la coálgebra descompone la lista en sublistas (elementos menores, pivote, elementos mayores)

```
c :: Ord a => [a] -> QsortF a [a]
c []     = NilF
c (x:xs) = ConsF smaller x larger
  where
    smaller = [y | y <- xs, y <  x]   -- Subproblema izquierdo
    larger  = [y | y <- xs, y >= x]   -- Subproblema derecho
```

La acción del álgebra combina las soluciones de los subproblemas concatenando las listas ya ordenadas

```
a :: QsortF a [a] -> [a]
a NilF = []
a (ConsF smaller p larger) = smaller ++ [p] ++ larger
```

Quicksort es un hylomorfismo entre la coálgebra c y el álgebra a

```
qsort :: Ord a => [a] -> [a]
qsort = a . fmap qsort . c
```

Limites de los hylomorfismos

La gran expresividad de los hylomorfismos tiene un costo, no toda hyloecuación tiene solución, mucho menos una solución única.

Podemos considerar solo aquellas (co)álgebras que garantizan una solución única para cada hyloecuación. Se conocen como **coálgebras recursivas**(F-$\mathbf{Rec}(\mathscr{C})$) y **álgebras corecursivas**(F-$\mathbf{Corec}(\mathscr{C})$).

Se puede probar que, bajo ciertas condiciones, los hylomorfismos siguen una relación de "adjunción":

$$\mathrm{Hylo}(L(C,\gamma),(A,\alpha)) \cong \mathrm{Hylo}((C,\gamma),R(A,\alpha))$$

Los funtores L y R preservan la recursividad y corecursividad respectivamente:

$$L : G\text{-}\mathbf{Rec}(\mathscr{D}) \to F\text{-}\mathbf{Rec}(\mathscr{C})$$

$$R : F\text{-}\mathbf{Corec}(\mathscr{C}) \to G\text{-}\mathbf{Corec}(\mathscr{D})$$