Hylomorfismos conjugados

Basado en ...

1. Motivación

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

```
data List a = [] | 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

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)

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
```

Usar el patrón *fold* nos permite reutilizar código, mejorar la legibilidad y facilitar el razonamiento sobre los programas. Además, cualquier optimización o mejora en la implementación de `foldr` se verá reflejada automáticamente en todos los algoritmos que lo utilizan.

No todos los algoritmos recursivos siguen el patrón *fold*. Por ejemplo, *makeTree* no consume ninguna estructura de datos para devolver su resultado y si bien *qsort* consume una estructura de tipo lista, previamente realiza otras acciones por lo que no puede implementarse directamente como un *fold*.

Sin embargo, muchos de los algoritmos de recursión estructurada pueden unificarse bajo un mismo esquema general, los *hylomorfismos*.

F-algebras y F-coálgebras

Definiciones

Dada una categoría $\mathcal{C}$ y un endofuntor F: $\mathcal{C} \to \mathcal{C}$ denominado *funtor base*, una F-álgebra es un par (A,) donde A es un objeto de $\mathcal{C}$ denominado *carrier* y : F A $\to$ A es una morfismo en $\mathcal{C}$ denominado *acción*. Cuando el contexto lo permite, me referiré a una F-álgebra particular solo por su acción.

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

```
F A  ----F f---->  F B
 |                  |
|                  |
 v                  v
 A ----f---------> B
```

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

De manera dual, una F-coálgebra es un par (A,) donde A es un objeto de $\mathcal{C}$ y : A $\to$ F A es un morfismo en $\mathcal{C}$. Un morfismo entre dos F-coálgebras (A,) y (B,) es un morfismo f: A $\to$ B en $\mathcal{C}$ tal que el siguiente diagrama conmuta:

```
 A ----f---------> B
 |                  |
|                  |
 v                  v
F A  ----F f---->  F B
```

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

F-álgebras iniciales y F-coálgebras terminales

Al objeto inicial de la categoría de F-álgebras, si existe, se le denomina *álgebra inicial* y se denota como $(\mu F, in_F)$. De manera dual, al objeto terminal de la categoría de F-coálgebras, si existe, se le denomina *coálgebra terminal* y se denota como $(\nu F, out_F)$.

Lema de Lambek Sea $(\mu F, in_F)$ un álgebra inicial. Entonces, $in_F : F(\mu F) \to \mu F$ es un isomorfismo.

Corolario Sea $(\nu F, out_F)$ una coálgebra terminal. Entonces, $out_F : \nu F \to F(\nu F)$ es un isomorfismo.

Punto fijo de un funtor Un objeto X de una categoría $\mathcal{C}$ es un *punto fijo* del endofuntor $F : \mathcal{C} \to \mathcal{C}$ si $F(X) \cong X$.

Teorema Los carriers de un álgebra inicial y una coálgebra terminal son puntos fijos del endofuntor F.

Al único morfismo de F-álgebras de $(\mu F, in_F)$ a cualquier otra F-álgebra (A, α) se le denomina *catamorfismo* o simplemente *fold* y se denota como $(|\alpha|) : \mu F \to A$(*alpha* esta rodeado por **banana brackets**). De manera dual, al único morfismo de F-coálgebras de cualquier otra F-coálgebra (A, α) a $(\nu F, out_F)$ se le denomina *anamorfismo* y se denota como $|(\alpha)| : A \to \nu F$(no pude encontrar el simbolo de parentesis, son como dos pilares ovalados).

Estructuras de datos como F-álgebras y F-coálgebras

Muchas estructuras de datos pueden definirse como F-álgebras o F-coálgebras donde el *carrier* es un punto fijo de F.

Ejemplo: Listas finitas de tipo a

Podemos ver los tipos de un lenguaje como objetos de una categoría y a las funciones que operan entre ellos como los homorfismos de dicha categoría. Consideremos la categoría **Hask** cuyos objetos son tipos de Haskell y cuyos morfismos son las funciones **totales** entre dichos tipos.

Las listas en Haskell se definen como

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

Las listas de este tipo son un punto fijo del endofuntor `ListF a`:

```haskell
data ListF a x = NilF | ConsF a x
  deriving (Show,Eq, Functor)
-- deriving Functor genera automáticamente una función fmap para ListF
-- fmap:: (x->y) -> ListF a x -> ListF a y
-- Que no es más que el mapeo de ListF a sobre morfismos
```

```
-- Notar que List a  ListF a (List a)
```

Podemos definir un acción sobre `List a`:

```
inn :: ListF a (List a) -> List a
inn NilF = []
inn (ConsF x xs) = x:xs
-- alpha es una acción y en particular también es un isomorfismo
```

En particular $(\mathrm{List}\, A, \alpha)$ es un álgebra inicial, asi que es posible definir un catamorfismo desde `List a` hacia cualquier otro carrier a partir del inverso de la acción `inn`:

```
innInv :: List a -> ListF a (List a)
innInv [] = NilF
innInv (x:xs) = ConsF x xs

-- Necesitamos saber a que f-algebra ir, usamos la acción f del F-algebra destino para sabe
cata :: (ListF a b -> b) -> List a -> b
cata f = a . fmap (cata f) . innInv

-- Podemos usar cata para generalizar foldr:
foldr:: (a->b->b) -> b -> List a -> b
foldr op e = cata alg
  where
    alg NilF = e
    alg (ConsF x xs) = op x xs
```

Ejemplo: Arboles binarios posiblemente infinitos

El tipo:

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

Es un punto fijo del endofuntor `TreeF a`:

```
data TreeF a x = E_F | L_F a | N_F x a x
  deriving (Show,Eq, Functor)
```

Podemos definir una acción sobre `Tree a`:

```
out :: Tree a -> TreeF a (Tree a)
out E = E_F
out (L x) = L_F x
out (N l x r) = N_F l x r
```

Que constituye una coálgebra terminal y para cualquier otra $TreeF$-coálgebra (C, c) existe un anamorfismo desde A hacia `Tree a`:

```haskell
outInv :: TreeF a (Tree a) -> Tree a
outInv E_F = E
outInv (L_F x) = L x
outInv (N_F l x r) = N l x r

ana :: (b -> TreeF a b) -> b -> Tree a
ana f = outInv . fmap (ana f) . f

-- Podemos usar ana para generalizar la generación de árboles:

-- Generar un arbol binario completo de altura n con etiquetas n
hasta :: Int -> Tree Int
hasta n = ana gen n
  where
    gen x
      | x==0 = E_F
      | x==1 = L_F x
      | otherwise = N_F (x-1) x (x-1)

-- Generar un arbol binario infinito con etiquetas n
infinito:: a -> Tree a
infinito x = ana gen x
  where
    gen x = N_F x x x
```

Los arboles posiblemente infinitos cuya ramificación es determinada por un funtor G y que toman etiquetas en A constituyen una coálgebra especial denominada $Cofree_G\ A$

Hylomorfismos

A grandes rasgos, dado un endofuntor F sobre una categoría $\mathcal{C}$ tenemos que: - El endofuntor F puede dar una cierta "estructura recursiva" a los objetos de $\mathcal{C}$. - El mismo endofuntor 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. 2. Se resuelve cada uno de los subproblemas. 3. Se combinan las soluciones de los subproblemas para obtener la solución del problema original.

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

Definición

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

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

Es decir, un hylomorfismo hace conmutar el siguiente diagrama:

```
      C ----h---------> A
      |                 |
     |                 |
      v                 v
    F C --F h------> F A
```

El esquema "Divide & Conquer" se puede interpretar de la siguiente manera: - La coálgebra (C, γ) descompone el problema original en subproblemas más pequeños. - El morfismo $F\ h$ resuelve cada uno de los subproblemas. - La álgebra (A, α) combina las soluciones de los subproblemas para obtener la solución del problema original.

Los catamorfismos y anamorfismos son hylomorfismos Sea $(\mu F, in)$ el álgebra inicial y (A, α) una F-álgebra cualquiera. El catamorfismo $(\!|\alpha|\!) : \mu F \to A$ satisface la siguiente ecuación:

$$(\!|\alpha|\!) \circ in = \alpha \circ F\ (\!|\alpha|\!)$$

in es un isomorfismo por lo que se puede reordenar la ecuación anterior para obtener una hyloecuación:

$$(\!|\alpha|\!) = \alpha \circ F\ (\!|\alpha|\!) \circ in^{-1}$$

Dado que $in^{-1} : \mu F \to F(\mu F)$, in^{-1} define una F-coálgebra. Por lo tanto, $(\!|\alpha|\!)$ es la solución de una hyloecuación y por ende es un hylomorfismo.

De manera analoga se puede probar que un anamorfismo es un hylomorfismo.

Ejemplo: Quicksort El algoritmo de ordenamiento rápido (quicksort) se puede definir como un hylomorfismo.

```
-- Considerar el endofuntor QsortF definido como:
data QsortF a x = Nil | Cons x a x
  deriving Functor
-- fmap:: (a->b) -> QsortF c a -> QsortF c b
-- fmap Nil = NIl
-- fmap (Cons l p r) = Cons (f l) p (f r)

-- La acción de la coálgebra genera los subproblemas
```

```haskell
c :: Ord a => [a] -> QsortF a [a]
c []     = Nil
c (x:xs) = Cons smaller x larger
  where
    smaller = [y | y <- xs, y <  x]
    larger  = [y | y <- xs, y >= x]

-- La acción del álgebra combina las soluciones de los subproblemas
a :: QsortF a [a] -> [a]
a Nil = []
a (Cons smaller p larger) = smaller ++ [p] ++ larger

-- Qsort es un hylomorfismo entre la coálgebra c y el álgebra a
qsort :: Ord a => [a] -> [a]
qsort = a . fmap qsort . c
```

Ejemplo: Recursión de cola El funtor $(A + -)$ puede usarse para modelar la recursión de cola. Sea A un conjunto fijo, una función recursiva de cola puede retornar con un valor A o bien puede continuar a una siguiente iteración. Los programas que usan recursión de cola son capturados por la siguiente hyloecuación:

$$h = (id\ id) \circ (A + h) \circ c = (id\ h) \circ c$$

```haskell
-- Considerar Either A B como el coproducto A+B
data Either a x = Left a | Right x
  deriving (Show,Eq)
-- fmap f (Left a) = Left a
-- fmap f (Right x) = Right (f x)

-- Accion de la coálgebra
c :: a -> Either A a
c x = if some_condition
        then Left value_of_type_A --- termina la recursión
        else Right next_iteration_value --- continua la recursión

-- La acción del álgebra simplemente retorna el valor de tipo A
a :: Either A A -> A
a (Left a) = a
a (Right b) = b

tailRecursion :: a -> A
tailRecursion = a . fmap tailRecursion . c
```

Coálgebras recursivas, álgebras corecursivas y reglas de unicidad

Los Hylomorfismos son altamente expresivos, en el sentido de que la enorme mayoría de los esquemas de recursión estructurada pueden definirse como hylomorfismos. Pero esta expresividad viene con un costo, no hay garantía de la existencia o unicidad de un hylomorfismo h entre una F-coálgebra (C, γ) y una F-álgebra (A, α) cualquiera.

En el ejemplo de recursión de cola, si la coalgebra c fuese definida como:

```
c x = Right x
```

Entonces el hylomorfismo `tailRecursion` generaria una función que diverge para cualquier entrada.

El problema es que la coálgebra puede llegar a generar una cantidad infinita de subproblemas mientras que el álgebra requiere que todos los subproblemas sean resueltos para poder combinar sus resultados. En este caso no existe un hylomorfismo entre ambas estructuras.

Para evitar estos problemas se pueden considerar aquellas coálgebras que para cualquier álgebra la hyloecuación tiene una única solución y, de manera dual, aquellas álgebras que para cualquier coálgebra la hyloecuación tiene una única solución. Las primeras reciben el nombre de *coálgebras recursivas* y las segundas el nombre de *álgebras corecursivas*.

Toda álgebra inicial es corecursiva y toda coálgebra terminal es recursiva: El hylomorfismo que resuelve la hyloecuación en estos casos es simplemente el catamorfismo o anamorfismo respectivamente.

Es de interés preguntarse si existen otras álgebras corecursivas o coálgebras recursivas además de las iniciales y terminales. Las reglas de unicidad nos permiten construir nuevas álgebras corecursivas y coálgebras recursivas a partir de otras ya conocidas.

Definiciones previas

Funtores entre álgebras

El funtor olvido Dado que una F-álgebra(y respectivamente una F-coálgebra) posee más estructura que la categoría $\mathcal{C}$ sobre la cual está definida, es posible definir un funtor olvido de manera similar al de su contraparte en **Set**. El funtor olvido de F-$\mathbf{Alg}(\mathcal{C})$ en $\mathcal{C}$ se denota como U_F y su análogo sobre G-coálgebras se denota como U^G. Cuando operan sobre objetos, ambos funtores simplemente retornan el *carrier* de la álgebra o coálgebra respectivamente. Cuando operan sobre morfismos, ambos funtores retornan el mismo morfismo en $\mathcal{C}$.

Funtores promocion(o lifting) Un funtor $\bar{H} : F\text{-}\mathbf{Alg}(\mathscr{C}) \to G\text{-}\mathbf{Alg}(\mathscr{C})$ es una *promoción* (o *lifting*) de un funtor $H : \mathscr{C} \to \mathscr{C}$ si el siguiente diagrama conmuta:

```
F-Alg(\mathscr{C})  ----\bar H---->  G-Alg(\mathscr{C})
        |                              |
      U_F                            U_G
        v                              v
    \mathscr{C}  ------H------->  \mathscr{C}
```

Los funtores promoción solo cambian acciones, los *carriers* y los morfismos permanecen fijos. Un funtor promoción especial puede ser definido a partir de una transformación natural $\lambda : G \circ H \to H \circ F$. De esta forma se define el funtor promoción H^{λ} como:

$$H^{\lambda}(A, \alpha) = (HA, H\alpha \circ \lambda_A) \quad H^{\lambda}(h) = h$$

De manera dual se pueden definir los funtores copromoción (o *colifting*) entre categorías de coálgebras. Un funtor $\bar{H} : F\text{-}\mathbf{Coalg}(\mathscr{C}) \to G\text{-}\mathbf{Coalg}(\mathscr{C})$ es una *copromoción* de un funtor $H : \mathscr{C} \to \mathscr{C}$ si el siguiente diagrama conmuta:

```
F-Coalg(\mathscr{C})  ----\bar H---->  G-Coalg(\mathscr{C})
        |                              |
      U^F                            U^G
        v                              v
    \mathscr{C}  ------H------->  \mathscr{C}
```

Y dada una transformación natural $\lambda : H \circ F \to G \circ H$, se define el funtor copromoción H_{λ} como:

$$H_{\lambda}(A, \alpha) = (HA, \lambda_A \circ H\alpha) \quad H_{\lambda}(h) = h$$

Adjunciones Dadas dos categorías $\mathscr{C}$ y $\mathscr{D}$ localmente pequeñas(como por ejemplo, **Hask**), la adjunción determinada por los funtores $L : \mathscr{C} \to \mathscr{D}$ y $R : \mathscr{D} \to \mathscr{C}$ con unidad de adjunción $\eta : 1_{\mathscr{C}} \to R \circ L$ y counidad de adjunción $\epsilon : L \circ R \to 1_{\mathscr{D}}$. La misma define un isomorfismo natural entre los conjuntos de morfismos:

$$\mathrm{Hom}_{\mathscr{D}}(LA, B) \cong \mathrm{Hom}_{\mathscr{C}}(A, RB)$$

Al isomorfismo que relaciona los morfismos $LC \to D$ lo denoto como $\lceil - \rceil$ y al isomorfismo que relaciona los morfismos $C \to RD$ lo denoto como $\lfloor - \rfloor$.

Transformaciones naturales conjugadas Las transformaciones naturales conjugadas surgen de la idea de estudiar como se relaciona una adjunción entre 2 categorías con otra adjunción entre otras 2 categorías a través de funtores que relacionan ambas parejas de categorías. De manera informal, sean las adjunciones $L \dashv R : \mathscr{C} \to \mathscr{D}$ y $L' \dashv R' : \mathscr{C}' \to \mathscr{D}'$ y dos funtores $H : \mathscr{C} \to \mathscr{C}'$

y $K : \mathcal{D} \to \mathcal{D}'$. Dos transformaciones naturales $\sigma : L' \circ K \to H \circ L$ y $\tau : K \circ R \to R' \circ H$ son *conjugadas* y se denota como $\sigma \dashv \tau$ si ambas estan relacionadas mediante adjunciones:

$$\lfloor Hf \circ \sigma_A \rfloor' = \tau_B \circ K \lfloor f \rfloor$$

o bien

$$H \lceil g \rceil \circ \sigma_A = \lceil \tau_B \circ Kg \rceil'$$

Para todo $f \in \mathrm{Hom}_{\mathcal{C}}(LA, B)$ y todo $g \in \mathrm{Hom}_{\mathcal{D}}(A, RB)$. Una propiedad importante es que es posible determinar σ si se conoce τ y viceversa.

Rolling rule

Ahora consideramos àlgebras y coálgebras definidas por la composición de dos endofuntores base. La *rolling rule* nos permite ...

Suponiendo que se tiene el siguiente diagrama, donde (L, R) forman una adjunción entre dos categorías $\mathcal{C}$ y $\mathcal{D}$:

```
(L.R)-Alg(\mathscr{C})  ------------------>  (R.L)-Alg(\mathscr{D})
      |                                         |
      |                                         |
      v                                         v
 \mathscr{C}  ------------------------->  \mathscr{D}


 (L.R)-Coalg(\mathscr{C})  <------------------  (R.L)-Coalg(\mathscr{D})
      |                                         |
      |                                         |
      v                                         v
 \mathscr{C}  <------------------------  \mathscr{D}
```