Hylomorfismos conjugados

Agustín Fernández Bergé

Dic 15, 2025

Hylomorfismos conjugados

Basado en el paper "Conjugate Hylomorphisms" de Hinze, Wu y Gibbons.

1. Motivación

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

```haskell
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:

```haskell
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:

```haskell
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*:

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

```haskell
# 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 (genera una estructura); y si bien `qsort` consume una estructura de tipo lista, previamente realiza otras acciones (descomposición en sublistas) 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**. Estos combinan la generación de estructuras (como `makeTree`) con su consumo (como `foldr`), capturando el paradigma "Divide & Conquer" de manera elegante.

2. F-álgebras y F-coálgebras

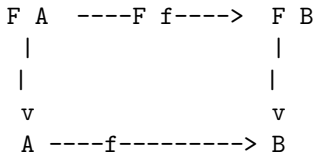
2.1 Definiciones

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

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

Intuitivamente, α especifica cómo "colapsar" o "evaluar" un nivel de estructura FA en un valor A. Cuando el contexto lo permite, nos referiremos a una F-álgebra particular solo por su acción.

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



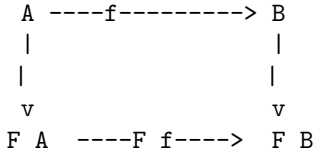
Las F-álgebras y sus morfismos forman una categoría denominada *categoría de F-álgebras* y denotada como F-**Alg**$(\mathcal{C})$.

De manera dual, una **F-coálgebra** es un par (A, α) donde:

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

Intuitivamente, α especifica cómo "generar" o "despliegar" un nivel de estructura FA a partir de un valor A.

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



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

2.2 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 $(\mu F, \text{in}_F)$. De manera dual, al objeto terminal de la categoría de F-coálgebras, si existe, se le denomina **coálgebra terminal**

y se denota como $(\nu F, \text{out}_F)$.

Lema de Lambek Sea $(\mu F, \text{in}_F)$ un álgebra inicial. Entonces, $\text{in}_F : F(\mu F) \to \mu F$ es un isomorfismo.

Intuición: La acción del álgebra inicial no sólo colapsa un nivel de estructura, sino que establece una correspondencia biunívoca entre $F(\mu F)$ y μF.

Corolario Sea $(\nu F, \text{out}_F)$ una coálgebra terminal. Entonces, $\text{out}_F : \nu F \to F(\nu F)$ 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 $(\mu F, \text{in}_F)$ 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 único morfismo de F-coálgebras de cualquier otra F-coálgebra (A, α) a $(\nu F, \text{out}_F)$ se le denomina **anamorfismo** o **unfold** y se denota como $|(\alpha)| : A \to \nu F$ (usando *lens brackets*).

2.3 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. Esta perspectiva permite:

- Separar la forma de la estructura (dada por F) de los datos que contiene
- Definir algoritmos genéricos que funcionen para cualquier estructura con la misma forma
- Razonar formalmente sobre propiedades de los algoritmos

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 es un punto fijo de ListF a x
```

Podemos definir un acción sobre `List a`:

```haskell
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 (List A, α) 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`:

```haskell
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 saberlo.
```

```haskell
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:

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

Es un punto fijo del endofuntor `TreeF a`:

```haskell
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`:

```haskell
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 $Cofree_G\ A$

3. 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 (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)

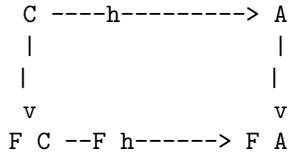
Podemos usar lo visto hasta ahora para definir este esquema de recursión de manera generalizada como un **hylomorfismo**.

3.1 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 homomorfismo de coálgebra a álgebra) si satisface la siguiente **hyloecuación**:

$$h = \alpha \circ Fh \circ \gamma$$

Es decir, un hylomorfismo hace conmutar el siguiente diagrama:



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

Interpretación del esquema "Divide & Conquer":

- La coálgebra (C, γ) **descompone** el problema original (de tipo C) en subproblemas más pequeños (dados por la estructura FC)
- El morfismo Fh **resuelve recursivamente** cada uno de los subproblemas, aplicando h a cada componente
- El álgebra (A, α) **combina** las soluciones de los subproblemas (dadas por FA) para obtener la solución del problema original (de tipo A)

3.2 Los catamorfismos y anamorfismos son hylomorfismos

Sea $(\mu F, \text{in})$ el álgebra inicial y (A, α) una F-álgebra cualquiera. El catamorfismo $(|\alpha|) : \mu F \to A$ satisface la siguiente ecuación:

$$(|\alpha|) \circ \text{in} = \alpha \circ F(|\alpha|)$$

in es un isomorfismo por el Lema de Lambek, por lo que se puede reordenar la ecuación anterior para obtener una hyloecuación:

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

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

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

Conclusión: Los catamorfismos y anamorfismos son casos especiales de hylomorfismos.

3.3 Ejemplo: Quicksort

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

```haskell
-- 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 _ NilF = NilF
-- fmap f (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
```

3.4 Ejemplo: Recursión de cola

El funtor $(A + -)$ (coproducto con un tipo fijo A) puede usarse para modelar la recursión de cola. Sea A un tipo fijo, una función recursiva de cola puede:

- Retornar con un valor de tipo A (caso base)
- Continuar a una siguiente iteración (caso recursivo)

Los programas que usan recursión de cola son capturados por la siguiente hyloecuación:

$$h = (\mathrm{id} \nabla \mathrm{id}) \circ (A + h) \circ c = (\mathrm{id} \nabla h) \circ c$$

Donde ∇ denota el cotupling (combinar dos funciones en un coproducto).

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

-- Acción de la coálgebra: decide si terminar o continuar
c :: a -> Either A a
c x = if some_condition
        then Left value_of_type_A     -- Termina la recursión (caso base)
        else Right next_iteration_value -- Continúa la recursión (caso recursivo)

-- La acción del álgebra simplemente retorna el valor de tipo A
-- o el resultado de la siguiente iteración
a :: Either A A -> A
```

```
a (Left a) = a   -- Caso base: retornar el valor
a (Right b) = b  -- Caso recursivo: ya se evaluó recursivamente

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

4. Coálgebras recursivas y álgebras corecursivas

Los hylomorfismos son altamente expresivos: 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 coálgebra c fuese definida como:

```
c x = Right x  -- Siempre continúa, nunca termina
```

Entonces el hylomorfismo `tailRecursion` generaría una función que **diverge** (nunca termina) para cualquier entrada.

El problema fundamental: 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, consideramos:

- **Coálgebras recursivas**: aquellas coálgebras que para cualquier álgebra la hyloecuación tiene una única solución
- **Álgebras corecursivas**: aquellas álgebras que para cualquier coálgebra la hyloecuación tiene una única solución (de manera dual)

En estos casos, el único hylomorfismo entre un álgebra corecursiva α y una coálgebra γ se denota como $(|\alpha \leftarrow \gamma|)$, y en el caso dual se denota como $|(\alpha \leftarrow \gamma)|$.

Teorema: 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.

5. Reglas de unicidad

5.2 Rolling rule

Definiciones previas

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-**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 $\mathcal{C}$.

Intuición: El funtor olvido "olvida" la estructura algebraica, recordando solo el carrier subyacente.

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

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

```
  U_F                          U_G
   v                            v
\mathscr{C}  ------H-------> \mathscr{D}
```

Intuición: Los funtores promoción "elevan" un funtor entre categorías a un funtor entre categorías de álgebras. Solo cambian acciones, los *carriers* y los morfismos permanecen fijos (en el sentido de que $U_G \circ \bar{H} = H \circ U_F$).

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:

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

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

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

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

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

La Rolling rule

Ahora consideramos álgebras y coálgebras definidas por la **composición de dos endofuntores** base.

Supongamos que tenemos el siguiente diagrama, donde (L, R) son dos funtores entre dos categorías $\mathcal{C}$ y $\mathcal{D}$:

```
\begin{tikzcd}
(L\circ R)-Alg(\mathscr{C}) \arrow[d, "U^{L \circ R}"'] \arrow[rrr, "\bar R"] &  &  & (R\circ L)-Alg(\maths
\mathscr{C} \arrow[rrr, "R"', shift right]                                     &  &  & \mathscr{D} \arrow[l
(L\circ R)-CoAlg(\mathscr{C}) \arrow[u, "U_{L\circ R}"]                         &  &  & (R\circ L)-CoAlg(\ma
\end{tikzcd}
```

Se puede construir $\bar{R}$ y $\bar{L}$ a partir de una transformación natural $\lambda : L \circ (R \circ L) \to (L \circ R) \circ L$; una transformación que cumple esto es la identidad: $\bar{L} = L_{id}$ y $\bar{R} = R^{id}$.

La rolling rule establece una "adjunción" entre dos tipos de hylomorfismos:

Teorema (Rolling rule) Sea (A, α) una $(L \circ R)$-álgebra en $\mathcal{C}$ y (C, γ) una $(R \circ L)$-coálgebra en $\mathcal{D}$. Entonces existe una correspondencia biunívoca:

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

Nota: La relación es de "adjunción" entre comillas porque los hylomorfismos no forman una categoría (dos hylomorfismos no se pueden componer en general).

Consecuencia: La rolling rule permite conseguir coálgebras recursivas y álgebras corecursivas a partir de otras ya conocidas.

Teorema (Preservación de recursividad y corecursividad) Las copromociones preservan recursividad y las promociones preservan corecursividad:

$$\underline{L} : (R \circ L)\text{-}\mathbf{Rec}(\mathcal{D}) \to (L \circ R)\text{-}\mathbf{Rec}(\mathcal{C})$$

$$\overline{R} : (L \circ R)\text{-}\mathbf{Corec}(\mathcal{C}) \to (R \circ L)\text{-}\mathbf{Corec}(\mathcal{D})$$

Donde **Rec** denota la clase de coálgebras recursivas y **Corec** denota la clase de álgebras corecursivas.

5.3 Conjugate rule

La rolling rule solo puede aplicarse cuando el funtor base es la composición de dos funtores. La **conjugate rule** extiende esta idea a cualquier par de endofuntores $F : \mathcal{C} \to \mathcal{C}$ y $G : \mathcal{D} \to \mathcal{D}$ cuando existe una adjunción entre $\mathcal{C}$ y $\mathcal{D}$ y dos transformaciones naturales especiales.

Definiciones previas

Adjunciones Dadas dos categorías $\mathcal{C}$ y $\mathcal{D}$ localmente pequeñas (como por ejemplo **Hask**), una **adjunción** determinada por los funtores $L : \mathcal{C} \to \mathcal{D}$ y $R : \mathcal{D} \to \mathcal{C}$ con unidad de adjunción $\eta : 1_{\mathcal{C}} \to R \circ L$ y counidad de adjunción $\varepsilon : L \circ R \to 1_{\mathcal{D}}$, se denota como $L \dashv R$.

La adjunción define un isomorfismo natural entre los conjuntos de morfismos:

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

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

Transformaciones naturales conjugadas Las transformaciones naturales conjugadas surgen de la idea de estudiar cómo se relaciona una adjunción entre 2 categorías con otra adjunción entre otras 2 categorías a través de funtores que relacionan ambas parejas de categorías.

Sean las adjunciones $L \dashv R : \mathcal{C} \to \mathcal{D}$ y $L' \dashv R' : \mathcal{C}' \to \mathcal{D}'$ y dos funtores $H : \mathcal{C} \to \mathcal{C}'$ 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 están relacionadas mediante adjunciones:

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

o equivalentemente:

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

Para todo $f \in \mathrm{Hom}_{\mathcal{D}}(LA, B)$ y todo $g \in \mathrm{Hom}_{\mathcal{C}}(A, RB)$.

Propiedad importante: Es posible determinar σ si se conoce τ y viceversa. Esta relación biunívoca es fundamental para las reglas de unicidad.

La conjugate rule

Definición Sean $F : \mathcal{C} \to \mathcal{C}$ y $G : \mathcal{D} \to \mathcal{D}$ dos endofuntores, supongamos que existe:

- Una adjunción $L \dashv R : \mathcal{C} \to \mathcal{D}$
- Dos transformaciones naturales conjugadas $\sigma : L \circ G \to F \circ L$ y $\tau : G \circ R \to R \circ F$

En este caso, los funtores L y R se pueden promocionar hacia las categorías de álgebras y coálgebras utilizando las transformaciones naturales conjugadas. Se tiene el siguiente diagrama:

```
\begin{tikzcd}
F-Alg(\mathscr{C}) \arrow[rr, "R^\tau"] \arrow[d, "U^F"']                                                    &      & F-Al
\mathscr C \arrow["F"', loop, distance=2em, in=215, out=145] \arrow[rr, "R"', shift right=2] & \bot & \mat
F-Coalg(\mathscr{C}) \arrow[u, "U_F"]                                                                        &      & G-Co
\end{tikzcd}
```

Podemos definir una "adjunción" entre hylomorfismos de manera similar a la rolling rule:

Teorema (Conjugate rule) Sea (A, α) una F-álgebra en $\mathcal{C}$ y (C, γ) una G-coálgebra en $\mathcal{D}$. Entonces existe una correspondencia biunívoca:

$$\mathrm{Hylo}(L_\sigma(C, \gamma), (A, \alpha)) \cong \mathrm{Hylo}((C, \gamma), R^\tau(A, \alpha))$$

Consecuencia: De manera similar a la rolling rule, se pueden conseguir coálgebras recursivas y álgebras corecursivas a partir de otras ya conocidas:

Teorema (Preservación con transformaciones conjugadas) Las copromociones preservan recursividad y las promociones preservan corecursividad si las mismas se realizan usando transformaciones naturales conjugadas:

$$L_\sigma : G\text{-}\mathbf{Rec}(\mathcal{D}) \to F\text{-}\mathbf{Rec}(\mathcal{C})$$

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

La conjugate rule indica que hay una correspondencia biunívoca entre un par de hylomorfismos denominados **hylomorfismos conjugados**. Usando la notación de hylomorfismos únicos se puede escribir como sigue:

$$\lfloor (|\alpha \leftarrow L_\sigma \gamma|) \rfloor = (|R^\tau \alpha \leftarrow \gamma|)$$

$$\lceil |(R^\tau \alpha \leftarrow \gamma)| \rceil = |(\alpha \leftarrow L_\sigma \gamma)|$$

5.4 Ejemplo: Hylo-shift Law

Usando la conjugate rule es posible derivar nuevas propiedades y esquemas de recursión a partir de una adjunción y un par de transformaciones naturales conjugadas.

Sea (A, α) una F-álgebra, (C, γ) una G-coálgebra sobre una categoría $\mathcal{C}$, y sea $\eta : G \to F$ una transformación natural.

Una adjunción sobre $\mathcal{C}$ se puede formar a partir del funtor identidad $(\mathrm{Id} \dashv \mathrm{Id})$. La misma transformación η induce un par conjugado de transformaciones naturales: - $\eta : \mathrm{Id} \circ G \to F \circ \mathrm{Id}$ - $\eta : G \circ \mathrm{Id} \to \mathrm{Id} \circ F$

Por lo que la conjugate rule induce los hylomorfismos conjugados:

$$(|\alpha \circ \eta_A \leftarrow \gamma|) = (|\alpha \leftarrow \eta_C \circ \gamma|)$$

Esta ley se conoce como **hylo-shift law** y permite "mover" una transformación natural entre la acción de un álgebra y la acción de una coálgebra dentro de un hylomorfismo.

5.5 Ejemplo: Mutu-Hylos

Elegir como adjunción los funtores $(\Delta \dashv (\times))$ (donde Δ es el funtor diagonal y $(\times)$ es el producto) da como resultado el esquema de recursión **mutu-hylos**, donde cada álgebra requiere del resultado de la otra para consumir su estructura.

Un ejemplo del patrón mutu-hylo es el juego Minimax:

```haskell
-- Minimax: Dos jugadores comienzan en la raíz de un árbol finito. En cada
-- turno pueden elegir si ir a la rama izquierda o la derecha del árbol.
-- El puntaje final es la suma de los valores de los nodos visitados.
-- Un jugador siempre trata de maximizar el puntaje mientras otro trata
-- de minimizarlo. ¿Cuál es el puntaje final?

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

-- Álgebra que maximiza: toma el máximo de las dos ramas
-- (usa el resultado del minimizador en las sublistas)
a1 :: (Num p, Ord p) => TreeF p (a, p) -> p
a1 E_F = 0
a1 (N_F l v r) = v + (snd l `max` snd r)  -- snd l es el puntaje del minimizador

-- Álgebra que minimiza: toma el mínimo de las dos ramas
-- (usa el resultado del maximizador en las sublistas)
a2 :: (Num p, Ord p) => TreeF p (p, b) -> p
a2 E_F = 0
a2 (N_F l v r) = v + (fst l `min` fst r)  -- fst l es el puntaje del maximizador
```

```haskell
data Tree a = E | N (Tree a) a (Tree a)
  deriving Show

split :: (a->b) -> (a->c) -> (a->(b,c))
split f g x = (f x, g x)

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

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

-- Mutu-hylomorfismo: ambas funciones se llaman mutuamente
maximize :: (Num a, Ord a) => Tree a -> a
maximize = a1 . fmap (split maximize minimize) . out

minimize :: (Num a, Ord a) => Tree a -> a
minimize = a2 . fmap (split maximize minimize) . out

-- Ejemplo de uso:
-- someTree :: Tree Int
-- someTree = N (N E 2 E) 3 (N E 4 E)
-- maximize someTree == 7
-- minimize someTree == 5
```