Resumen T1

Tomás Felipe Melli

March 31, 2025

Índice

1 Esquemas de recursión sobre listas

1.1 Recursión Estructural

La **Recursión Estructural** es un esquema recursivo en el que se cumplen 2 propiedades fundamentales:

1. **El caso base devuelve un valor fijo z.**
2. El caso recursivo se escribe usando (cero, una o muchas veces) x y (g xs), pero sin usar el valor de xs ni otros llamados recursivos. En otras palabras, cuando decimos que **no se usa directamente xs**, estamos diciendo que **no estamos haciendo ninguna operación explícita sobre el contenido de xs en el cuerpo de la recursión.** No estamos accediendo o manipulando directamente los elementos de xs, solo estamos pasando xs a la llamada recursiva (g xs). Esto garantiza que cada llamada recursiva se hace con una lista más pequeña, y eventualmente la recursión termina.

Sea `g :: [a] -> b` definida por dos ecuaciones:

```
g []       = ⟨caso base⟩
g (x : xs) = ⟨caso recursivo⟩
```

g está dada por **recursión estructural** si:

1. El caso base devuelve un valor fijo z.
2. El caso recursivo se escribe usando (cero, una o muchas veces) x y (g xs), pero sin usar el valor de xs ni otros llamados recursivos.

```
g []       = z
g (x : xs) = ... x ... (g xs) ...
```

Algunos ejemplos :

```haskell
suma :: [Int] -> Int
suma [] = 0
suma (x : xs) = x + suma xs

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

-- Insertion sort
isort :: Ord a => [a] -> [a]
isort [] = []
isort (x : xs) = insertar x (isort xs)
```

Qué pasa con ...

```haskell
-- Selection sort
ssort :: Ord a => [a] -> [a]
ssort [] = []
ssort (x : xs) = minimo (x : xs) : ssort (sacarMinimo (x : xs))
```

Es recursión estructural ? La respuesta es que no. El caso recursivo está escrito de manera que se realiza una operación sobre el contenido de xs.

1.1.1 foldr

La función **foldr abstrae el esquema de recursión estructural** :

```haskell
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x : xs) = f x (foldr f z xs)
```

Decimos que **toda recursión estructural es una instancia de foldr.** Esto es verdadero ya que foldr toma : una función binaria, un valor y una lista. De forma que captura el mismo patrón recursivo de procesar los elementos de una lista, aplicando una función de forma recursiva sobre los elementos y utilizando un valor base para el caso base. Miremos estos ejemplos :

```haskell
suma :: [Int] -> Int
suma = foldr (+) 0
```

$$
\begin{aligned}
& suma[1,2] \\
&= foldr\ (+)\ 0\ [1,2] \\
&= (+)\ 1\ (foldr\ (+)\ 0\ [2]) \\
&= (+)\ 1\ ((+)\ 2\ (foldr\ (+)\ 0\ [\,])) \\
&= (+)\ 1\ ((+)\ 2\ 0) \\
&= (+)\ 1\ 2 \\
&= 3
\end{aligned}
$$

Vale para un producto u operaciones booleanas como :

```
producto :: [Int] -> Int
producto = foldr (*) 1

and, or :: [Bool] -> Bool
and = foldr (&&) True
or = foldr (||) False
```

1.1.2 reverse con foldr

Reverse básicamente invierte el orden de una lista, para ello :

```
reverse :: [a] -> [a]
reverse [] = []
reverse (x : xs) = reverse xs ++ [x]
```

Como es una función que sigue el esquema de recursión estructural, la podemos reescribir como instancia de foldr.

```
reverse = foldr (\ x rec -> rec ++ [x]) []
```

En esta λ, recibe **rec** que es el resultado acumulado de la recursión y **x** que es el elemento actual de la lista.

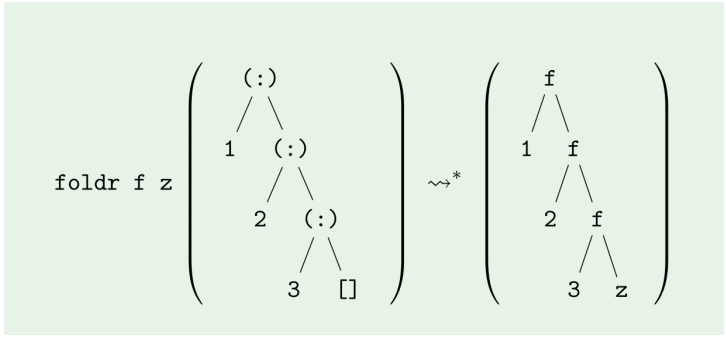
$$
\begin{aligned}
& reverse\ [1,2,3] \\
&= foldr\ (\ x\ rec \to rec\ +\!+\ [x])\ [\,]\ [1,2,3] \\
&= (\ x\ rec \to rec\ +\!+\ [x])\ 1\ (foldr\ (\ x\ rec \to rec\ +\!+\ [x])\ [\,]\ [2,3]) \\
&= (\ x\ rec \to rec\ +\!+\ [x])\ 1\ ((\ x\ rec \to rec\ +\!+\ [x])\ 2\ (foldr\ (\ x\ rec \to rec\ +\!+\ [x])\ [\,]\ [3])) \\
&= (\ x\ rec \to rec\ +\!+\ [x])\ 1\ ((\ x\ rec \to rec\ +\!+\ [x])\ 2\ ((\ x\ rec \to rec\ +\!+\ [x])\ 3\ (foldr\ (\ x\ rec \to rec\ +\!+\ [x])\ [\,]\ [\,]))) \\
&= (\ x\ rec \to rec\ +\!+\ [x])\ 1\ ((\ x\ rec \to rec\ +\!+\ [x])\ 2\ ((\ x\ rec \to rec\ +\!+\ [x])\ 3\ [\,])) \\
&= (\ x\ rec \to rec\ +\!+\ [x])\ 1\ ((\ x\ rec \to rec\ +\!+\ [x])\ 2\ [3]) \\
&= (\ x\ rec \to rec\ +\!+\ [x])\ 1\ [3,2] \\
&= [3,2]\ +\!+\ [1] \\
&= [3,2,1]
\end{aligned}
$$

Tenemos otras formas de implementar reverse con foldr :

```
reverse = foldr (\ x rec -> flip (++) [x] rec) []
reverse = foldr (\ x -> flip (++) [x]) []
reverse = foldr (\ x -> flip (++) ((: []) x)) []
reverse = foldr (\ x -> (flip (++) . (: [])) x) []
reverse = foldr (flip (++) . (: [])) []
```

1. Es análoga a la primera que desarrollamos pero usamos flip para invertir el orden de los parámetros
2. Esta es lo mismo que la anterior, pero **rec se define automáticamente** sin necesidad de escribirlo explícitamente.
3. En esta versión se usa **(:) cons** para hacer lo equivalente a **[x]**
4. En este caso, se componen funciones con **(.)**. O sea, primero se aplica **(: [])** **x** a x y luego se aplica **flip (++)** con los argumentos rec y x.
5. Lo mimso que la de antes pero con **x implícito**

1.1.3 Plegado de listas a derecha



En particular, se puede demostrar que ...

```haskell
foldr (:) [] = id
foldr ((:) . f) [] = map f
foldr (const (+ 1)) 0 = length
```

Recordar que **const**

```haskell
const :: a -> b -> a
const x _ = x
```

Independientemente de lo que sea el segundo argumento, el resultado siempre será el primero.

1.1.4 Casos degenerados

```haskell
-- Es recursi n estructural (no usa la cabeza)
length :: [a] -> Int
length [] = 0
length (_ : xs) = 1 + length xs

-- Es recursi n estructural (no usa el llamado recursivo sobre la cola):
head :: [a] -> a
head [] = error "No tiene cabeza."
head (x : _) = x
```

1.2 Recursión Primitiva

La **recursión primitiva** es un esquema recursivo definido por una función cuando cumple que :

1. **El caso base devuelve un valor fijo z.**
2. El caso recursivo se escribe usando (cero, una o muchas veces) x, (g xs) y también xs, pero sin hacer otros llamados recursivos.

Sea `g :: [a] -> b` definida por dos ecuaciones:

```
g []       = ⟨caso base⟩
g (x : xs) = ⟨caso recursivo⟩
```

Decimos que la definición de g está dada por **recursión primitiva** si:

1. El caso base devuelve un valor fijo `z`.
2. El caso recursivo se escribe usando (cero, una o muchas veces) `x`, `(g xs)` y también `xs`, pero sin hacer otros llamados recursivos.

```
g []       = z
g (x : xs) = ... x ... xs ... (g xs) ...
```

Similar a la recursión estructural, pero permite referirse a `xs`.

En simples términos, una función se llama a sí misma con un valor más simple del problema hasta llegar a un caso base.

```
trim :: String -> String
>> trim "Hola PLP"     "Hola PLP"
trim [] = []
trim (x : xs) = if x ==          then trim xs else x : xs
```

Se puede abstraer el esquema de la recursión primitiva con la función **recr**.

1.2.1 recr

Toda recursión primitiva es una instancia de recr.

```
recr :: (a -> [a] -> b -> b) -> b -> [a] -> b
recr f z [] = z
recr f z (x : xs) = f x xs (recr f z xs)
```

Miremos cómo escribir *trim* usando recr:

```
trim = recr (\ x xs rec -> if x ==          then rec else x : xs) []
```

1.3 Recursión Iterativa

Según este esquema, la función se definine por :

1. **El caso base devuelve el acumulador ac.**
2. El caso recursivo invoca inmediatamente a (g ac' xs), donde ac' es el acumulador actualizado en función de su valor anterior y el valor de x.

Miremos los ejemplos:

```
-- Reverse con acumulador.
reverse  :: [a] -> [a] -> [a]
reverse  ac [] = ac
reverse  ac (x : xs) = reverse  (x : ac) xs

-- Pasaje de binario a decimal con acumulador.
-- Precondici n: recibe una lista de 0s y 1s.
bin2dec  :: Int -> [Int] -> Int
bin2dec  ac [] = ac
bin2dec  ac (b : bs) = bin2dec  (b + 2 * ac) bs

-- Insertion sort con acumulador.
isort  :: Ord a => [a] -> [a] -> [a]
isort  ac [] = ac
isort  ac (x : xs) = isort  (insertar x ac) xs
```

En este esquema, la función que lo abstrae se llama **foldl**.

1.3.1 foldl

Toda recursión iterativa es una instancia de foldl.

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f ac [] = ac
foldl f ac (x : xs) = foldl f (f ac x) xs
```

Sea `g :: b -> [a] -> b` definida por dos ecuaciones:

$$\begin{aligned} \texttt{g ac []} &= \langle \textit{caso base} \rangle \\ \texttt{g ac (x : xs)} &= \langle \textit{caso recursivo} \rangle \end{aligned}$$

Recursión iterativa

Decimos que la definición de g está dada por *recursión iterativa* si:

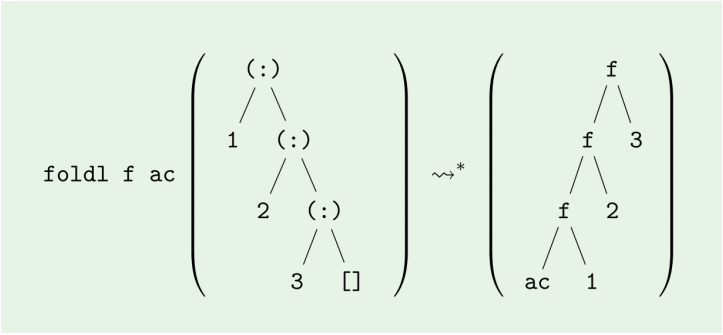
1. El caso base devuelve el acumulador `ac`.
2. El caso recursivo invoca inmediatamente a `(g ac' xs)`, donde `ac'` es el acumulador actualizado en función de su valor anterior y el valor de `x`.

La función foldl es un operador de iteración.

Algorithm 1 foldl

```
0: function FOLDL(f, ac, xs)
0:   while xs no es vacía do
0:     ac ← f(ac, head(xs))
0:     xs ← tail(xs)
0:   end while
0:   return ac
```

1.3.2 Plegado de listas a izquierda



foldr y foldl se comportan diferente.

```
foldr (   ) z [a, b, c] = a     (b     (c     z))
foldl (   ) z [a, b, c] = ((z     a)     b)     c
```

Las estrellitas representan la función binaria. En el caso en que esta estrellita sea un **operador asociativo y conmutativo**(es decir, una operación que cumpla con la propiedad de asociatividad y conmutatividad) como..

```
suma = foldr (+) 0 = foldl (+) 0
producto = foldr (*) 1 = foldl (*) 1
and = foldr (&&) True = foldl (&&) True
or = foldr (||) False = foldl (||) False
```

Entonces, definen la misma función.
Supongamos la función que realiza el pasaje de binario a decimal

```
bin2dec :: [Int] -> Int
bin2dec = foldl (\ ac b -> b + 2 * ac) 0
```

$$bin2dec\ [1,0,0]$$
$$foldl\ (ac\ b\ ->\ b + 2 * ac)\ 0$$
$$foldl\ (ac\ b\ ->\ b + 2 * ac)\ (1 + 0)$$
$$foldl\ (ac\ b\ ->\ b + 2 * ac)\ (0 + 2 * (1 + 0))$$
$$foldl\ (ac\ b\ ->\ b + 2 * ac)\ (0 + 2 * (0 + 2 * (1 + 0)))$$
$$(0 + 2 * (0 + 2 * (1 + 0)))$$
$$= 4$$

Se puede demostrar que

```
foldl (flip (:)) [] = reverse
```

2 Tipos de datos algebraicos

Existen los tipos de datos que conocemos como "primitivos" (Char, Int, Float,..) y a su vez se pueden definir nuevos con la **claúsula data**

```
data Tipo =    declaracin    de los constructores
```

Los tipos de datos algebraicos se dividen en dos clasificaciones, los **Tipos Sumatorios** y los **Tipos de producto**. Vamos a ver el ejemplo de Día que es de tipo enumerado es un tipo sumatorio muy simple que representa una disyunción de varias opciones, donde cada opción (constructor) no tiene parámetros.

2.0.1 Tipos Enumerados

En los **tipos enumerados**, el tipo puede tomar un número finito y fijo de valores, y esos valores se llaman **constructores**. Cada uno de estos constructores representa un valor posible de tipo Dia. Además de teenr **valores fijos**, son **no extensibles**, es decir, una vez que definís un tipo enumerado en Haskell, no podés agregar nuevos valores o constructores al tipo. Hagamos el ejemplo en ghci

```
data Dia = Dom | Lun | Mar | Mie | Jue | Vie | Sab
ghci> :{
ghci| esFinde :: Dia -> Bool
ghci| esFinde Sab = True
ghci| esFinde Dom = True
ghci| esFinde _ = False
ghci| :}
ghci> esFinde Sab
True
ghci> esFinde Dom
True
ghci> esFinde Lun
False
```

La primer línea define un **tipo algebraico llamado Dia** con **7 constructores: Dom, Lun, Mar, Mie, Jue, Vie, y Sab.** Cada uno de estos es un constructor de valor de tipo Dia, y no se pueden agregar más constructores.

2.0.2 Tipo Producto

Para este caso de tipo algebraico, representa una combinación de valores (en el ejemplo que vamos a ver, un nombre, apellido y edad) y tiene constructores con parámetros.

```
ghci> data Persona = LaPersona String String Int
ghci> :{
ghci| nombre :: Persona -> String
ghci| nombre (LaPersona n _ _ ) = n
ghci| :}
ghci> tom = LaPersona "Tomas" "Melli" 24
ghci> nombre tom
"Tomas"
```

En este caso, de tipo producto, se combinan distintos tipos de datos, y en este ejemplo, se define un sólo constructor.

2.0.3 Muchos constructores ... Muchos Parámetros

Por ejemplo, si definimos el tipo algebraico **Forma**

```
ghci> data Forma = Rectangulo Float Float | Circulo Float
ghci> :{
ghci| area :: Forma -> Float
ghci| area (Rectangulo ancho alto) = ancho * alto
ghci| area (Circulo radio) = radio * radio * pi
ghci| :}
ghci> cuadrado = Rectangulo 3.0 3.0
ghci> circulo = Circulo 4.0
ghci> area cuadrado
9.0
ghci> area circulo
50.265484
```

2.0.4 Constructores recursivos

```
data Nat = Zero | Succ Nat
```

Declara dos constructores, **Zero** y **Succ**.
Podemos construir los números naturales de manera recursiva, en este tipo se construyen de forma inductiva. El número 0 es Zero, y cualquier número mayor se obtiene aplicando el constructor Succ al número anterior. La recursión sobre el tipo Nat se puede entender mediante el principio de inducción estructural, lo que significa que para definir funciones que operen con Nat, basta con definir cómo se comporta la función para el caso base (cuando el valor es Zero) y cómo se comporta en el caso recursivo (cuando el valor es Succ de algún otro valor de tipo Nat). Dicho esto, los valores de tipo Nat tendrán la forma ...

```
Zero
Succ Zero
Succ (Succ Zero)
```

```
Succ (Succ (Succ Zero))
...
```

En una diapo se habla de que en Haskell se puede trabajar sobre estructuras infinitas y dice *"en Haskell las definiciones recursivas se interpretan de manera coinductiva en lugar de inductiva"*. Esto quiere decir que la inducción (en tipos recursivos) es el principio que se aplica cuando un tipo recursivo tiene una estructura finita o termina en algún momento. Y la coinducción, por otro lado, es el principio que se aplica cuando un tipo recursivo puede ser infinito o no tiene un caso base terminante. Los tipos coinductivos no requieren que la recursión termine, sino que permiten estructuras infinitas. Por ejemplo, los streams (secuencias infinitas de datos) son típicamente definidos de manera coinductiva.

2.0.5 Caso General

```
data T = CBase1      parmetros
       ...
       | CBase n      parmetros
       | CRecursivo 1      parmetros
       ...
       | CRecursivo m      parmetros
```

- Los **constructores base** no reciben parámetros de tipo **T**.
- Los **constructores recursivos** reciben *al menos un parámetro de tipo T*.
- Los valores de tipo T *son los que se pueden construir aplicando constructores base y recursivos* **un número finito de veces y sólo esos** (Entendemos la definición de T de forma inductiva).

Ejemplo del banco :

```
type Cuenta = String
data Banco = Iniciar
           | Depositar Cuenta Int Banco
           | Extraer Cuenta Int Banco
           | Transferir Cuenta Cuenta Int Banco

bancoPLP = Transferir "A" "B" 3 (Depositar "A" 10 Iniciar)

saldo :: Cuenta -> Banco -> Int
saldo cuenta Iniciar = 0
saldo cuenta (Depositar cuenta monto banco)
           | cuenta == cuenta = saldo cuenta banco + monto
           | otherwise = saldo cuenta banco

saldo cuenta (Extraer cuenta monto banco)
           | cuenta == cuenta = saldo cuenta banco - monto
           | otherwise = saldo cuenta banco

saldo cuenta (Transferir origen destino monto banco)
           | cuenta == origen = saldo cuenta banco - monto
           | cuenta == destino = saldo cuenta banco + monto
           | otherwise = saldo cuenta banco
```

Las listas son un caso particular de tipo algebraico :

```
data Lista a = Vacia | Cons a (Lista a)
```

O, con la notación ya conocida

```
data [a] = [] | a : [a]
```

Los árboles binarios son otro ejemplo:

```
data AB a = Nil | Bin (AB a) a (AB a)
```

Se pueden definir funciones sobre ellos como:

```
-- Visitar el nodo, luego el sub rbol izquierdo y luego el sub rbol derecho
preorder :: AB a -> [a]
preorder Nil = []
preorder (Bin izq x der) = [x] ++ preorder izq ++ preorder der

-- Visitar el sub rbol izquierdo, luego el sub rbol derecho y luego el nodo
postorder :: AB a -> [a]
postorder Nil = []
postorder (Bin izq x der) = postorder izq ++ postorder der ++ [x]
```

```haskell

-- Visitar el sub rbol izquierdo, luego el nodo y luego el sub rbol derecho
inorder :: AB a -> [a]
inorder Nil = []
inorder (Bin izq x der) = inorder izq ++ [x] ++ inorder der

-- Insertar
-- Pre: el  rbol  de entrada es un AB ordenado y sin repetidos.
-- Post: el  rbol  resultante es un AB ordenado y sin repetidos que
contiene a los elementos del AB de entrada y al elemento dado.
insertar :: Ord a => a -> AB a -> AB a
insertar x Nil = Bin Nil x Nil
insertar x (Bin izq y der)
        | x < y = Bin (insertar x izq) y der
        | x > y = Bin izq y (insertar x der)
        | otherwise = Bin izq y der
```

3 Esquema de Recursión sobre otras estructuras

3.1 Recursión Estructural

La recursión estructural se generaliza a tipos algebraicos en general. Supongamos que T es un tipo algebraico. Dada una función g :: T -¿ Y definida por ecuaciones:

```haskell
g (CBase1      parmetros     ) =   caso   base 1
...
g (CBase n      parmetros     ) =   caso   base n
g (CRecursivo 1      parmetros     ) =   caso   recursivo 1
...
g (CRecursivo m      parmetros     ) =   caso   recursivo m
```

Decimos que g está dado por *recursión estructural* si :

1. Cada caso base se escribe combinando los parámetros.
2. Cada caso recursivo se escribe combinando:
   - Los parámetros del constructor que no son de tipo T.
   - El llamado recursivo sobre cada parámetro de tipo T.

   Pero ...
   - Sin usar los parámetros del constructor que son de tipo T.
   - Sin hacer a otros llamados recursivos.

3.1.1 Ejemplo foldAB

```haskell
foldAB :: b -> (b -> a -> b -> b) -> AB a -> b
foldAB cNil cBin Nil = cNil
foldAB cNil cBin (Bin i r d) = cBin (foldAB cNil cBin i) r (foldAB cNil cBin d)
```

Es una forma genérica de recorrer un árbol binario y realizar una operación sobre sus elementos.

- **cNil** es el valor que se devuelve cuando el árbol es vacío (Nil).
- **cBin** es una función que toma tres argumentos: el resultado del recorrido del subárbol izquierdo, el valor del nodo actual y el resultado del recorrido del subárbol derecho. La función cBin se utiliza para combinar los resultados de los subárboles y el nodo en el caso de un nodo no vacío (Bin).

1. **Caso base**: cuando el árbol es vacío (Nil), simplemente se devuelve el valor cNil.
2. **Caso recursivo** : si el árbol no es vacío, es un nodo Bin con un subárbol izquierdo i, un valor r en el nodo y un subárbol derecho d. Primero se recurre al subárbol izquierdo con *foldAB cNil cBin i*. Luego se recurre al subárbol derecho con *foldAB cNil cBin d*. Finalmente, la función cBin se aplica a los tres elementos: el resultado del subárbol izquierdo, el valor del nodo r, y el resultado del subárbol derecho.

3.1.2 ¿Qué función es (foldAB Nil Bin)?

Es una función que recursivamente reconstruye el árbol, manteniendo la estructura de nodos y subárboles, y simplemente pasa a los subárboles izquierdo y derecho sin hacer ninguna operación adicional sobre los valores de los nodos.

3.1.3 Definir mapAB :: (a -¿ b) -¿ AB a -¿ AB b usando foldAB

La función mapAB aplica una función a todos los elementos de un árbol binario.

3.1.4 Definir maximo :: AB a -¿ Maybe a usando foldAB

La función maximo busca el valor máximo en un árbol binario. Si el árbol es vacío, retorna Nothing, y si el árbol tiene nodos, retorna el valor máximo encontrado.

3.1.5 Definir altura :: AB a -¿ Int usando foldAB

La función altura calcula la altura del árbol binario. La altura de un árbol vacío es 0, y la altura de un árbol con nodos es la altura del subárbol más grande más uno.