Paradigmas de Lenguajes de Programación

Tomás Agustín Hernández

Programación Funcional

Consiste en definir funciones y aplicarlas para procesar información.
Las funciones son verdaderamente funciones (parciales):

- Aplicar una función no tiene efectos secundarios.
- A una misma entrada le corresponde siempre la misma salida.
- Las estructuras de datos son inmutables.

Las funciones, además son datos como cualquier otro:

- Se pueden pasar como parámetros.
- Se pueden devolver como resultados.
- Pueden formar parte de estructuras de datos. Ej.: Un árbol binario que en sus nodos hay funciones.

Expresiones

Son secuencias de símbolos que sirven para representar datos, funciones, y funciones aplicadas a los datos.
Una expresión puede ser:

- Un constructor: True, False, [], (:), 0, 1, 2.
  - Type Constructor: Es un constructor que se utiliza para crear un nuevo tipo.
  - Data Constructor: Se utiliza para crear valores de ese tipo.
  - Ej.: $data\ Color\ =\ Rojo \mid Verde \mid Azul$. Azul es un **data constructor** pues nos permite crear valores del tipo Color mientras que el Type Constructor es Color.
  - Ej.: $data\ Complejo\ =\ C\ Float\ Float$. C es una función que recibe dos Float y es un constructor de Complejo.
- Una variable: longitud, ordenar, x, xs (+), (*).
- La aplicación de una expresión a otra: ordenar lista, not True, (+) 1.

Función Parcialmente Aplicada

Una función parcialmente aplicada es una función a las cuales se llama otra función pero no se le proporcionan todos los argumentos.
Ahora, es importante que para que esta función parcialmente aplicada tenga sentido se le manden todos los parámetros.
Es una especie de $a \to b$ Ej.:

```
add :: Int -> Int -> Int
add x y = x + y

add5 :: Int -> Int
add5 = add 5

add5 es una función que ejecuta la función parcial add pues le manda solo un parámetro y necesita dos.
```

Una buena pregunta entonces es ¿pero por qué es una función parcial add 5? Pues hay una función add5 que la implementa sin pasarle todos los parámetros directamente explícitamente.

```
add5 3 -> Este 3 llega como "y" a add

add5 = add 5
add = 5 + y
add = 5 + 3
add = 8
```

Otro ejemplo

```
const :: a -> b -> a
const x y = x

const (const 1) 2 -> const 1
El llamado de (const 1) es una función parcialmente aplicada porque no envía el valor de y, sin embargo,
    const (const 1) 2 no la aplica parcialmente porque le termina mandando los dos parámetros.
```

Aplicación de Expresiones

Es asociativa hacia la izquierda:

- $f\ x\ y \equiv\ (f\ x)\ y$
- $((((f\ a)\ b)\ c)\ d)$: Primero calcula el resultado que devuelve la expresión f enviando el valor de a. Nótese que la ídea seria que (f a) devuelva una expresión del tipo función pues luego le pasamos otro parámetro (b).

Construyendo una lista paso a paso con constructores

Ej.: ¿Como construimos la lista [1, 2] utilizando constructores?

- Lo primero que necesitamos, es un constructor de listas. Para eso tenemos la expresión (:). Recordando que la aplicación de expresiones es asociativo a izquierda.
- Veamos su tipo desde GHCI $(:)\ ::\ a -> [a] -> [a]$.
- Necesitamos enviarle un valor de tipo a, una lista y como resultado, la operación (:) devuelve una lista.
- Preguntemos lo siguiente: ¿Con ((:) 1) nos basta para agregar a una lista? No, porque no estamos cumpliendo el tipado del constructor. Si quisieramos una lista con solamente el 1 si bastaría, pero acá también queremos el 2.
- Entonces, comenzamos aplicando la expresión (:) con el número 2, y ahí sí enviamos como segundo parámetro una lista vacía (que da como resultado) una lista vacía.
- $(((:)\ 2)\ []) = [2]$
- Por último, ((:) 1) [2] también cumple el tipo pues nos quedaría ((:) 1) [2] = [1, 2]

¿Y si quisieramos construir 1, 2, 3, 4? Recordemos la asociatividad a la izquierda, pero a la hora de querer utilizar una expresión, hay que cumplir el tipado. Hasta que ese tipado no se cumpla, Haskell tratará de resolver la expresión más profunda para ver si reduciendo cumple el tipo.

$$((:)1)\ (((:)\ 4)(((:)\ 2)(((:)\ 3)\ [])))$$

Esto lo podemos ver como $a \to (b \to (c \to (d)))$ pues para conocer a, necesitamos construir la lista de la derecha, para conocer a b necesitamos la lista de la derecha, para conocer a c necesitamos la lista de d, y d inicializa la lista vacía. Esto es importantísimo porque claramente no podríamos usar (:) 1 sin una lista. Entonces Haskell evalúa todo lo de la derecha hasta que obtenga una lista. Si no se obtuviera una lista, sería inválido.

Polimorfismo

Sucede en aquellas expresiones que tienen más de un tipo.

- $id\ ::\ a \to a$
- $(:) :: a \to [a] \to [a]$
- $fst :: (a, b) \to a$
- $cargarStringArray :: String -> [] \to [String]$. No es una expresión Polimórfica.

Nota: Es importante que si tenemos 2 parámetros de tipo a significa que los dos parámetros deben ser de ese tipo. Si tenemos 2 parámetros uno de tipo a y uno de b podría suceder que sean diferentes pero puede que sean el mismo.

Modelo de Cómputo (cálculo de valores)

Dada una expresión, se computa su valor usando las ecuaciones siempre y cuando estén bien tipadas.
Importante: Que una expresión se cumpla el tipado, no significa que devuelva un valor.

- $sumarUno :: a -> a$: No falla nunca.
- $division :: a -> b$: Falla si b = 0. Esto nos demuestra que aunque los parámetros estén bien tipados, podemos tener indefiniciones.

¿Cómo está dado un programa en Funcional?

Un programa funcional está dado por un conjunto de **ecuaciones orientadas**. Se les llama de esta forma pues del **lado izquierdo está lo que define y del lado derecho la definición** (o expresión que produce el valor) Una ecuación e1 = e2 se interpreta desde dos puntos de vista

- **Denotacional**: Declara que e1 y e2 tienen el mismo significado.
  - "**Denotan** lo mismo"
- **Operacional**: Computar el valor de e1 se reduce a computar el valor de e2.
  - "**Operan** de la misma forma"

¿Cómo es el lado izquierdo de una ecuación orientada?

No es una expresión arbitraria. Debe ser una función aplicada a **patrones**.
Un patrón puede ser:

- Una variable: a, b, c.
- Un comodin: _.
- Un constructor aplicado a patrones: Recordemos que un constructor sería algo que construye un tipo.
  - True, False, 1, etc.

Importante: El lado izquierdo NO debe contener variables repetidas.
Ej.: $iguales\ x\ x\ =\ True$ es una expresión mal formada. Esto pues dos valores que pueden ser diferentes, no pueden caer en la misma variable.
Ej.: $predecesor\ (n+1)\ =\ n$ también está mal formada porque estamos haciendo un cálculo del lado izquierdo, y recordemos que del lado izquierdo solo hay definiciones. Del lado derecho se hacen los cómputos o cálculo de los valores.

¿Cómo evalúamos las expresiones?

- Buscamos la subexpresión más externa que coincida con el lado izquierdo de una ecuación.
- Reemplazar la subexpresión que coincide con el lado izquierdo de la ecuación por la expresión correspondiente al lado derecho.
- Continuar evaluando la expresión resultante.

¿Cuando se detiene la evaluación de una expresión?

- Cuando el programa estalla.
  - Loop infinito.
  - Indefinición.
- Cuando la expresión es una función **parcialmente aplicada**. ¿que seria esto? ¿se refiere a utilizar mal la función parcialmente aplicada?
- La expresión es un constructor o un constructor aplicado. Ej.: True, (:) 1, [1, 2, 3].
  - Yo lo veo como algo más del tipo: una fórmula atómica o algo irreducible.

¿Cómo ayuda el Lazy Evaluation (Evaluación Perezosa) a Haskell?

Muchas veces nos ayuda a evitar tocar con un valor indefinido aunque esté ahí. Como no evalúa cosas que no necesita, si hay un indefinido por ahí y no necesita ni siquiera llegar, no lo toca.

```
1 indefinido :: Int
2 indefinido = indefinido
3 head(tail [indefinido, 1, indefinido])
```

¿Qué hace tail? Toma la cola de la lista, entonces como resultado arroja [1, indefinido] (ni siquiera evaluó el primer indefinido)
¿Qué hace head ahora entonces? [1] (ni siquiera evaluó el indefinido del final)

Importancia del orden de las Ecuaciones

El criterio más fuerte debe ir por encima del resto. Porque puede haber un caso que matchee y nunca se evalúe el siguiente caso.
Veamos un ejemplo:

```
esCorta (_:_:_) = False
esCorta _ = True
```

Considerando este orden:

- Si mando una lista que siempre tiene ≥ 3 elementos da False.
- Si mando una lista que tiene < 3 elementos es siempre True.

Cambiemos el orden y veamos qué cambia

```
esCorta _ = True
esCorta (_:_:_) = False
```

Considerando este orden:

- Si la lista tiene 0, 1, 2, 3, 4, ... n elementos siempre dará True.

En la segunda aplicación ¡nunca caemos en el caso 2! y esto es importante porque sabemos que si tiene 1 cae siempre en la primera.

Funciones de Orden Superior

Son funciones que reciben como parámetro otras funciones.
Definamos la composición de funciones $(g \ . \ f)$
Recordemos que en Álgebra vimos Composición de Funciones y es algo así: sean $A : B \to C$ y $D : A \to B$

La composición gof es $A \to B$. Es decir, va desde el dominio de D hasta la imagen de A. (la salida de una función debe ser la entrada de la otra.)

En Haskell, la podemos definir así

```
ent: Entrada sal: Salida

(.) :: (b -> c) -> (a -> b) -> (a -> c)

Queremos: (g . f) x => g (f x)

Desglosemos
(.) :: (ent. de g -> sal. de g) -> (ent. de f -> sal. de f) -> (ent. de f -> sal. de g)

El resultado de la composición nos da una nueva función que tiene como entrada un valor de tipo a, y la
    salida es un valor de tipo c.

Entonces, primero se evalúa f x, lo cual produce un resultado de tipo b.

Luego, este resultado (de tipo b) se pasa a g, produciendo un resultado de tipo c.

(g . f) x: envía el parámetro x a la función f, y el resultado de f x se manda a g. Esto da como resultado
    g (f x)
```

Otra forma de definir la composición es usando Notación Lambda.

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
g . f = \ x -> g (f x)
```

Nótese que la notación lambda es súper útil para definir funciones sin nombre, que solamente hagan algo. Esto es muy útil cuando queremos mandar una función por parámetro y que una función dada la llame a esta función.

¿Para qué queremos funciones de orden superior? Pt 1

```
dobleL :: [Float] -> [Float]
dobleL [] = []
dobleL (x:xs) = x * 2 : dobleL xs

esParL :: [Int] -> [Bool]
esParL [] = []
esParL (x:xs) = x `mod` 2 == 0 : esParL xs
```

¿Qué es lo que tienen en común? Todas tienen una estructura bastante similar.

```
g [] = []
g (x : xs) = f x : g xs
```

Lo único que cambia es **qué se hace** en cada paso recursivo.
Hagamos una pregunta ¿la cantidad de elementos de la entrada es igual a la de la salida? en este caso sí pero ¿qué operación se está haciendo? se están haciendo ciertas manipulaciones de los datos y se devuelve la información modificada en una lista nueva.
Esto, en varios lenguajes se conoce como **map**. Se hace una manipulación de los datos pero se devuelve **la misma cantidad de elementos**.

Map

Recibe como parámetro una función que es aplicada a todos los elementos y devuelve una lista nueva. Se utiliza para modificar los valores de una lista dada según lo que haga la función.

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

Entonces, podemos definir **qué operación de modificación** se le realizan a los elementos de una lista dada.

```
multiplicarPorDos :: [a] -> [b]
multiplicarPorDos xs = map (\x -> x * 2) xs

dividirPorDos :: [a] -> [b]
dividirPorDos xs :: map(\x -> x / 2) xs
```

Entonces gracias a Map nos abstraemos de tener miles de funciones con la misma estructura pero solo cambien **qué hacen** con los elementos.

¿Para qué queremos funciones de orden superior? Pt 2

```
negativos :: [Int] -> [Int]
negativos [] = []
negativos (x:xs) = if x < 0
                   then x : negativos xs
                   else negativos xs

pares :: [Int] -> [Int]
pares [] = []
pares (x:xs) = if x `mod` 2 == 0
               then x : pares xs
               else pares xs
```

¿Qué es lo que tienen en común? Todas tienen una estructura bastante similar, pero lo único que cambia es **cuando vamos a agregar a la lista los elementos**.

```
g [] = []
g (x:xs) = f x : g xs
```

Hagamos una pregunta ¿la cantidad de elementos de la entrada es igual a la de la salida? Puede que sí, puede que no.
¿Qué operación se está haciendo? se están filtrando ciertos elementos de una lista que no cumplan una condición dada.
Esto, en varios lenguajes se conoce como **filter**. Se devuelve una nueva lista con los elementos que cumplan una condición dada.

Filter

Recibe como parámetro una función que es aplicada a todos los elementos. Se utiliza para "borrar.elementos de una lista, o quitar aquellos que no cumplan un criterio dado.

```
filter :: (a -> Bool) -> [a] -> [b]
filter p [] = []
filter p (x : xs) = if p x
                    then x : filter p xs
                    else filter p xs
```

La función que le envíamos es para especificar **qué elementos nos queremos quedar**.
Entonces, podemos definir **qué operación de filtro** se le realizan a los elementos de una lista dada.

```
eliminarImpares :: [a] -> [b]
eliminarImpares xs = filter (\x -> x `mod` 2 == 0) xs

borrarNegativos :: [a] -> [b]
borrarNegativos xs = filter(\x -> x > 0) xs
```

Entonces gracias a Filter nos abstraemos de tener miles de funciones con la misma estructura pero solo cambien **con qué elementos nos quedamos** dada una condición

Recordando Haskell

Para ejecutar un archivo hay que instalar GHCI. Una vez instalado, nos paramos en la terminal en el directorio donde está el archivo que queremos ejecutar.

- Cargar archivo: :l nombreArchivo
- Ver tipo: :type tipo
- Ejecutar funcion: funcion parametro1 parametro2...
- Recargar archivo: :r
- Si necesitamos hacer cálculos para mandar un parámetro, usar paréntesis: Ej.: otherwise = n * factorial(n-1)

Maybe

El Maybe se utiliza en Haskell para recibir/devolver respuestas condicionales que pueden ser de un tipo u otro.

Se define como $data\ Maybe\ a\ =\ Nothing\ |\ Just\ a$

Ej.: $devolverFalsoSiVerdadero\ :\ Bool \rightarrow Prelude.Maybe\ Bool$

El Maybe deja la puerta abierta a un valor posible "Nothing". Entonces tenemos dos casos: Si me envian un True devuelvo False (tipo bool), caso contrario, devuelvo Nothing.

Either

El Either se utiliza en Haskell para poder recibir/devolver un parámetro que podría ser de un tipo u otro.
Se define como $data\ Either\ a\ b\ =\ Left\ a\ |\ Right\ b$

Para poder saber qué operación hacer según el tipo literalmente en código usamos (Left valor) o (Right valor).

Ej.: $devolverRepresentacionIntBool\ ::\ Either\ Int\ Bool\ \rightarrow\ Int$

Si es un entero, devuelvo ese mismo entero porque no hago nada. Eso lo hacemos con $Left(a)\ =\ a$, ahora, si el tipo es booleano tengo que decir explícitamente la respuesta según su valor. Es decir, $Right(False)\ =\ 0$ sino, $Right(True)\ =\ 1$.

Declaración de tipos en Haskell

Se utiliza $data\ nombretipo\ tipo\ =\ Tipo\ 1\ |\ Tipo\ 2$ El | se interpreta como .º bien"

Árboles Binarios

Es un tipo (para mi parecer) meramente recursivo.
data AB a = Nil | Bin (AB a) a (AB a) Nótese que es algo re contra recursivo, porque para definir el tipo de AB a decimos que es un Bin que a su vez es de AB a y a su vez AB a es otro árbol binario. Veamos unos ejemplos de esto

- Bin (Nil) Nil (Nil): es el árbol que no tiene ni siquiera raíz. Y nótese que en cada paréntesis es importante indicar el Nil pues es la forma de que el tipado de Haskell nos lo acepte.
- Bin (Bin Nil 3 Nil) 4 (Bin Nil 6 Nil): Es el árbol que comienza con un Nodo raíz que tiene el valor de 4. El hijo izquierdo del Nodo con valor 4 es otro árbol binario que tiene como valor 3 en su nodo y no tiene hijos. El hijo derecho del Nodo con valor 4 es otro árbol binario que tiene como valor 6 en su Nodo y no tiene hijos.

Y así sucesivamente, veamos un dibujo para tener algo más visual.
El siguiente árbol binario: Bin (Bin (Bin Nil 2 Nil) 3 Nil) 4 (Bin (Bin Nil 5 Nil) 6 Nil) representa el siguiente: