

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 | Verde | 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 $a \rightarrow b$ Ej.:

```
1 | add :: Int -> Int -> Int
2 | add x y = x + y
3 |
4 | add5 :: Int -> Int
5 | add5 = add 5
6 |
7 | 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.

```
1 | add5 3 -> Este 3 llega como "y" a add
2 |
3 | add5 = add 5
4 | add = 5 + y
5 | add = 5 + 3
6 | add = 8
```

Otro ejemplo

```
1 | const :: a -> b -> a
2 | const x y = x
3 |
4 | const (const 1) 2 -> const 1
5 | 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 idea sería 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 $(:) :: a \rightarrow [a] \rightarrow [a]$.
- 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 quisiéramos 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 quisiéramos 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 $a \rightarrow (b \rightarrow (c \rightarrow (d)))$ 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.

- $id :: a \rightarrow a$
- $(:) :: a \rightarrow [a] \rightarrow [a]$
- $fst :: (a, b) \rightarrow a$
- $cargarStringArray :: String \rightarrow [] \rightarrow [String]$. 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.

- $sumarUno :: a \rightarrow a$: No falla nunca.
- $division :: a \rightarrow b$: 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 comodín: $_$.
- 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 cálculos o cálculo de los valores.

¿Cómo evaluamos 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.

¿Cuándo se detiene la evaluación de una expresión?

- Cuando el programa estalla.
 - Loop infinito.
 - Indefinición.
- Cuando la expresión es una función **parcialmente aplicada**. ¿que sería 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 $[1, \text{indefinido}]$ (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:

```
1 | esCorta (_:_:_) = False
2 | 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.

Cambiamos el orden y veamos qué cambia

```
1 | esCorta _ = True
2 | 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

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 → Prelude.Maybe Bool*

El Maybe deja la puerta abierta a un valor posible "Nothing". Entonces tenemos dos casos: Si me envían 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 → 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:

