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.

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 $a \rightarrow (b \rightarrow (c \rightarrow (d)))$ 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.

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 $data\ Maybe\ a\ =\ Nothing \mid Just\ a$

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 $data\ Either\ a\ b\ =\ Left\ a \mid Right\ b$

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

Árboles Binarios

Es un tipo (para mi parecer) meramente recursivo.
$data\ AB\ a\ =\ Nil \mid Bin\ (AB\ a)\ a\ (AB\ a)$ 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: