PROGRAMANDO EN HASKELL

Capítulo 3 - Tipos y Clases

¿Qué es un Tipo?

Un tipo es un nombre para una colección de valores relacionados. Por ejemplo, en Haskell el tipo básico

```
Bool
```

contiene los dos valores lógicos:

```
False
```

```
True
```

Errores de Tipo

Se llama error de tipo al hecho de aplicar una función a uno o más argumentos del tipo equivocado.

```
> 1 + False
error ...
```

1 es un número y False es un valor lógico, pero + requiere dos números.

Tipos en Haskell

- Si el evaluar una expresión e producirá un valor de tipo t, entonces e tiene tipo t, escrito

```
e :: t
```

- Toda expresión válida tiene un tipo, el cual puede ser automáticamente calculado durante la compilación usando un proceso llamado inferencia de tipo.

- Todos los errores de tipo son encontrados en la compilación, lo que hace a los programas más seguros y más rápidos al quitar la necesidad de validar los tipos en tiempo de ejecución.

- En GHCi, el comando :type calcula el tipo de una expresión, sin evaluarla:

```
> not False
True

> :type not False
not False :: Bool
```

Tipos Básicos

Haskell tiene varios tipos básicos, incluyendo:

- `Bool` - valores lógicos
- `Char` - un solo carácter
- `String` - cadena de caracteres
- `Int` - números enteros
- `Float` - números de punto flotante

Tipos Lista

Una lista es una secuencia de valores del mismo tipo:

```
[False,True,False] :: [Bool]

['a','b','c','d'] :: [Char]
```

En general:

[t] es el tipo de listas con elementos de tipo t.

Nota:

- El tipo de una lista no dice nada sobre su tamaño:

```
[False,True] :: [Bool]

[False,True,False] :: [Bool]
```

- El tipo de elementos no está restringido. Ejemplo, podemos tener listas de listas:

```
[['a'],['b','c']] :: [[Char]]
```

Tipos Tupla

Una tupla es una secuencia de valores de diferente tipo:

```
(False,True) :: (Bool,Bool)

(False,'a',True) :: (Bool,Char,Bool)
```

En general:

(t1,t2,…,tn) es el tipo de n-tuplas cuyo componente en la posición i tiene tipo ti para toda i en 1…n.

Nota:

- El tipo de una tupla encodifica su tamaño:

```
(False,True) :: (Bool,Bool)

(False,True,False) :: (Bool,Bool,Bool)
```

- El tipo de componentes no está restringido:

```
('a',(False,'b')) :: (Char,(Bool,Char))

(True,['a','b']) :: (Bool,[Char])
```

Tipos Función

Una función es un mapeo de valores de un tipo a valores de otro tipo:

```
not :: Bool → Bool

even :: Int → Bool
```

En general:

t1 → t2 es el tipo de funciones que mapean valores de tipo t1 a valores de tipo t2.

Nota:

- La flecha $\rightarrow$ se escribe en el teclado como ->.

- Los tipos de argumento y resultado son irrestrictos. Ejemplo, las funciones con múltiples argumentos o resultados son posibles usando listas o tuplas:

```
add :: (Int,Int) → Int
add (x,y) = x+y

zeroto :: Int → [Int]
zeroto n = [0..n]
```

Funciones Currificadas

Es posible definir funciones con múltiples argumentos al regresar funciones como resultados:

```
add' :: Int → (Int → Int)
add' x y = x+y
```

add' toma un entero 'x', y regresa una función add' x. Esta función toma un entero 'y', y regresa el resultado x+y.

Nota:

- add y add' producen el mismo resultado final, pero add toma sus dos argumentos al mismo tiempo, en cambio add' los toma uno a la vez:

```
add :: (Int,Int) → Int

add' :: Int → (Int → Int)
```

- Las funciones que toman sus argumentos uno a la vez se llaman funciones currificadas, celebrando la obra de Haskell Curry con tales funciones.

- Las funciones con más de dos argumentos pueden ser currificadas, si regresamos funciones anidadas:

```
mult :: Int → (Int → (Int → Int))
mult x y z = x*y*z
```

mult toma un entero 'x' y regresa una función mult x, la cual a su vez toma un entero 'y' y regresa una función mult x y, que finalmente toma un entero 'z' y regresa el resultado x*y*z.

¿Por qué es útil Currificar?

Las funciones currificadas son mas flexibles que las funciones sobre tuplas, porque algunas veces pueden construirse funciones útiles al aplicar parcialmente una funcion currificada.

Ejemplo:

```
add' 1 :: Int → Int

take 5 :: [Int] → [Int]

drop 5 :: [Int] → [Int]
```

Convenciones de Currificado

Se adoptan dos convenciones simples para evitar el uso excesivo de paréntesis al utilizar funciones currificadas:

- La flecha $\rightarrow$ se asocia a la derecha.

$$\texttt{Int} \rightarrow \texttt{Int} \rightarrow \texttt{Int} \rightarrow \texttt{Int}$$

Significa $\texttt{Int} \rightarrow (\texttt{Int} \rightarrow (\texttt{Int} \rightarrow \texttt{Int}))$.

- Como consecuencia, es natural que la aplicación de funciones se asocie a la izquierda.

```
mult x y z
```

A menos que el uso de tuplas sea requerido explícitamente, todas las funciones en Haskell estan normalmente definidas de forma currificada.

Funciones Polimórficas

Una función es polimófica (“de muchas formas”) si su tipo contiene una o más variables de tipo.

```
length :: [a] → Int
```

Para todo tipo a, length toma una lista de valores de tipo a y regresa un entero.

Nota:

- Las variables de tipo pueden ser instanciadas a tipos diferentes en circunstancias diferentes:

```
> length [False,True]
2

> length [1,2,3,4]
4
```

a = Bool

a = Int

- Las variables de tipo deben comenzar con una letra minúscula, usualmente llamadas a, b, c, etc.

- Muchas de las funciones definidas en la librería estándar (Prelude) son polimórficas. Ejemplo:

```
fst :: (a,b) → a

head :: [a] → a

take :: Int → [a] → [a]

zip :: [a] → [b] → [(a,b)]

id :: a → a
```

Funciones Sobrecargadas

Una función polimórfica se dice sobrecargada si su tipo contiene una o más reestriccines de clase.

```
(+) :: Num a ⇒ a → a → a
```

Para cualquier tipo numérico a, (+) toma dos valores de tipo a y regresa un valor de tipo a.

Nota:

- Los tipos de variables reestringidos pueden ser instanciados a cualesquier tipo o tipos que satisfagan las reestricciones:

```
> 1 + 2
3

> 1.0 + 2.0
3.0

> 'a' + 'b'
ERROR
```

a = Int

a = Float

Char no es un tipo numérico

- Haskell tiene varias clases de tipo, incluyendo:

`Num`	-	Tipos Numéricos
`Eq`	-	Tipos de Igualdad
`Ord`	-	Tipos Ordenados

- Ejemplo:

```
(+)  :: Num a ⇒ a → a → a
(==) :: Eq a  ⇒ a → a → Bool
(<)  :: Ord a ⇒ a → a → Bool
```

Pistas y Tips

- Al definir una nueva función en Haskell, es útil comenzar escribiendo su tipo;

- En un script, es buena práctica especificar el tipo de cada nueva función definida;

- Al especificar los tipos de funciones polifmórficas que usan números, igualdad u ordenamiento, verifica la inclusion de las reestricciones de clase necesarias.

Ejercicios

(1) ¿Cuál es el tipo de cada uno de los siguientes valores?

```
['a','b','c']

('a','b','c')

[(False,'0'),(True,'1')]

([False,True],['0','1'])

[tail,init,reverse]
```

(2) ¿Cuál es el tipo de cada una de las siguientes funciones?

```
second xs = head (tail xs)

swap (x,y) = (y,x)

pair x y = (x,y)

double x = x*2

palindrome xs = reverse xs == xs

twice f x = f (f x)
```

(3) Verifica tus respuestas usando GHCi.

Fuente.

Profesor Graham Hutton.
http://www.cs.nott.ac.uk/~pszgmh/

Traducción al español.

Caleb Josue Ruiz Torres.
https://www.calebjosue.com