

Introducción a Haskell

Y a la programación funcional

Pablo Baeyens
@pbaeyens

Mario Román
@M42

OSL 2015

Índice

Haskell

Tipos

Funciones

Ejemplos

Más

¡Contribuye!

El código fuente de estas diapositivas está disponible en:

github.com/M42/osl-talk-haskell

Erratas, correcciones y aportaciones son bienvenidas.

Instalando Haskell Platform

El paquete `haskell-platform` contiene el compilador, depurador, gestor de librerías y otras utilidades para programar en Haskell. En otras distribuciones puede instalarse directamente `ghc` (Glasgow Haskell Compiler):

```
sudo apt-get install haskell-platform
```

Viene con un gestor de librerías: `cabal`

Sin efectos secundarios

En los lenguajes de programación funcional, una función toma un argumento y devuelve una salida. No altera el mundo alrededor ni cambia el valor de los argumentos.

En lenguajes imperativos, cuando llamamos a una función puede cambiar nuestras variables o escribir por pantalla. Eso hace que el orden de llamada de las funciones importe.

```
int n = 0;  
int next_n() { return n++; }  
next_n(); // n = 1
```

El intérprete: GHCi

GHC es un compilador de Haskell con GHCi como intérprete asociado. El intérprete permite los siguientes comandos:

- ▶ `:q` Quitar
- ▶ `:l` Cargar módulo
- ▶ `:r` Recargar módulos
- ▶ `:t` Consultar tipos

Una vez cargado el intérprete podemos utilizarlo para probar el lenguaje. Haskell permite operaciones aritméticas básicas, y operaciones con cadenas, listas o booleanos.

El intérprete: GHCi

Podemos probar el uso de un puñado de funciones simples. Las funciones se escriben dejando sus argumentos a su lado y separados por espacios. ¡Estamos usando **notación polaca**!

```
ghci> 3 + 4
7
ghci> (+) 2 9
11
ghci> succ 27
28
ghci> max 23 34
34
```

Tipos

Haskell tiene los tipos básicos ya construidos. Existen `Int`, `Bool`, `Char`, ...

```
ghci> :t True
True :: Bool
ghci> :t 'a'
'a' :: Char
ghci> :t "Una_string!"
"Una_string!" :: [Char]
ghci> :t not
not :: Bool -> Bool
```


Tipos

Los tipos de Haskell son **fuertes** y **estáticos**. La mayor parte de los errores se detectan en compilación como errores de tipo.

Además son **inferidos**, por lo que no tenemos por qué especificar el tipo de nuestras funciones:

```
ghci> let nand a b = not (a && b)
ghci> :t nand
nand :: Bool → Bool → Bool
```

Clases de tipos

Las clases de tipos agrupan a tipos con la misma interfaz. Por ejemplo, la clase `Eq`, define la función `==`.

```
2 :: Num a => a
pi :: Floating a => a
(==) :: Eq a => a -> a -> Bool
```

Las instancias de `Num` pueden sumarse y multiplicarse, las de `Show` convertirse a `String` y sobre las de `Integral` pueden calcularse restos modulares.

Variables de tipo

Haskell infiere siempre el tipo más general posible. Cuando algo puede tener varios tipos, coloca una **variable de tipo**, que puede ser luego sustituida.

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

Y la variable puede verse restringida por pertenecer a una clase.

Tipos algebraicos

Creamos nuevos tipos definiendo **constructores de datos**:
funciones que devuelven valores del tipo que definimos.

```
data () = () — Tipo void
data Bool = False | True
data Point = Point Float Float
data Triangle = Triangle Point Point Point
```

Podemos definir tipos que dependan de otros o tipos con constructores sin argumentos, como `Bool`.

Constructores de tipos

Los constructores de tipos son funciones sobre tipos: toman un tipo y devuelven otro.

Por ejemplo, `[]` construye listas, y `Maybe` construye un tipo que puede tener o no un valor.

```
"Haskell!" :: [Char]
[1,2,3,4]  :: (Num a) => [a]
[True, False, False] :: [Bool]
[] :: [a]
Just True  :: Maybe Bool
Nothing   :: Maybe a
```

Constructores de tipos

Sus definiciones son:

```
data [a]      = [] | a:[a]  
data Maybe a = Nothing | Just a
```

En las **listas**, el primer constructor es la lista vacía y el segundo antepone un elemento a otra lista.

En el caso de **Maybe** podemos tener algo de tipo `a` (`Just a`) o nada (`Nothing`).

Reconocimiento de patrones

Para definir una función sobre un tipo, definimos su comportamiento para cada constructor de datos del tipo:

```
neg :: Bool → Bool
neg False = True
neg True  = False
```

Podemos utilizar variables para sustituir cualquier argumento del constructor:

```
suma (Point a b) (Point c d) = Point (a+c) (b+d)
opuesto (Point a b) = Point (-a) (-b)
resta a b = suma a (opuesto b)
```

Recursividad

El tipo lista y las definiciones recursivas son la base de los programas de Haskell. Por ejemplo, para calcular la **longitud de una lista**, definimos la función `len` para sus dos constructores:

```
len []      = 0
len (_:xs) = 1 + len xs
```

El primer constructor nos proporciona el caso base y el segundo la ecuación recursiva.

Currificación

¿Por qué el tipo de `(+)`, por ejemplo, es `(Num a) => a -> a -> a` en lugar de ser `(Num a) => (a,a) -> a`?

Eso nos permite aplicar parcialmente una función. El tipo hay que leerlo realmente como `a -> (a -> a)`, es decir, al darle un número nos devuelve otra función:

```
(+)      :: Num a => a -> a -> a
(+3)     :: Num a => a -> a
(+3) 5   :: Num a => a
```

Es lo mismo decir `(+3) 5` que `(+) 3 5`.

Funciones de orden superior

Son funciones que toman funciones como argumento.

`map` toma una función y devuelve su versión sobre listas, elemento a elemento:

```
map :: (a → b) → ([a] → [b])
```

Ejemplo: `map not [True, True, False]`

`foldr` toma una función y un acumulador y aplica todos los elementos de la lista contra el acumulador.

```
foldr :: (a → a → a) → a → [a] → a
```

Ejemplo: `foldr (*) 1 [2,3,5,7]`

Especialización

Ahora, a partir de funciones tan generales, podemos crear las funciones básicas:

```
negation = map not
lowerText = map toLower
sum = foldr (+) 0
prod = foldr (*) 1
concat = foldr (++) []
and = foldr (&&) True
```

Ejemplo: `lowerText "aBcDEfG"`

Quicksort

Implementación del algoritmo Quicksort

```
qsort []      = []  
qsort (x:xs) = qsort [y | y<-xs, y<=x]  
              ++ [x]  
              ++ qsort [y | y<-xs, y>x]
```

Árboles binarios

Los definimos como vacíos o un nodo con dos árboles:

```
data Tree a = Empty
             | Node a (Tree a) (Tree a)
```

Ejemplo: `Node 4 (Node 3 Empty Empty) Empty`

Y esto nos deja definir funciones sobre ellos fácilmente:

```
preorder :: Tree a → [a]
preorder Empty = []
preorder Node x a b = x : (preorder a ++ preorder b)
```

Curry-Howard

Como las instancias sólo pueden construirse desde los constructores definidos y las funciones no tienen efectos secundarios, podemos razonar fácilmente la corrección del código. Usamos inducción estructural:

```
qsort []      = []  
qsort (x:xs) = qsort [y | y<-xs, y<=x]  
              ++ [x]  
              ++ qsort [y | y<-xs, y>x]
```

Por ejemplo, quicksort funciona porque ordena correctamente una lista vacía y porque, supuesto que funcione para listas menores que una dada, funciona para ella.