Introducción a Haskell

Mario Román *Pablo Baeyens*

El compilador y el intérprete

`ghc` es el compilador más utilizado para Haskell. Contiene gran cantidad de extensiones y es actualmente el compilador más utilizado para este lenguaje.

`ghci` es el intérprete interactivo asociado a `ghc`. Para simplificar el proceso de E/S en esta introducción usaremos el intérprete para cargar un archivo [1] y trabajaremos los archivos desde el propio intérprete. El intérprete permite los siguientes comandos:

- `:q` Salir
- `:l` Cargar módulo
- `:r` Recargar módulos
- `:t` Comprobar tipos [2]
- `:i` Obtener información

Una vez instalado `ghci`, crea un archivo `archivo.hs` y cárgalo con `:l archivo`.

INSTALACIÓN DE GHC

Haskell Platform contiene un compilador, depurador, gestor de librerías y otras utilidades para programar en Haskell.
Puede instalarse desde su página oficial o desde los repositorios de la mayoría de distribuciones GNU/Linux (`haskell-platform`).

[1] Existe una versión online del intérprete interactivo en ghc.io, aunque éste no permite cargar archivos.

[2] Si queremos que `ghci` nos muestre el tipo de las expresiones por defecto podemos usar `:set +t`.

Conceptos básicos

Haskell es un lenguaje de programación funcional puro, perezoso y con tipos fuertes y estáticos, nombrado en honor a Haskell Curry.

En Haskell las funciones **no tienen efectos secundarios**: no alteran el mundo ni cambian el valor de sus argumentos. No existen las variables: la ejecución de un programa se basa en evaluar expresiones: sustituir *iguales por iguales*.

Estas restricciones permiten optimizar enormemente nuestros programas y paralelizar la evaluación.

Al no haber efectos secundarios el orden de evaluación no importa permitiendo una evaluación **perezosa**, y pudiendo razonar algebraicamente sobre nuestros programas.

EFECTOS SECUNDARIOS

¿Podemos optimizar $f(x) - f(x) \to 0$?

En un lenguaje imperativo las funciones pueden tener efectos secundarios: si su valor de retorno depende del mundo exterior o es aleatorio las expresiones podrían no ser equivalentes.

Un lenguaje funcional nos da más libertad: la ausencia de efectos secundarios permite realizar gran cantidad de optimizaciones y razonamientos.

Llamada a funciones

Haskell permite operaciones aritméticas básicas, y operaciones con cadenas, listas o booleanos. Las siguientes líneas pueden ejecutarse en el intérprete, devolviendo el resultado esperado:

```
5 + 6 * 4
(+) 5 6
"Hello␣" ++ "World!"
True && False
not True
"Hello" == "Hello"
5 == (+) 2 3
```

Si ejecutamos algo como `True == 'a'` Haskell nos indicará que los tipos no concuerdan:

```
Couldn't match expected type 'Bool' with actual type '
    Char'
In the second argument of '(==)', namely ''a''
In the expression: True == 'a'
```

Las mayoría de las funciones son prefijas. Se escriben dejando sus argumentos a su lado y separados por espacios. No encerramos los argumentos entre paréntesis como se haría en la mayoría de lenguajes:

```
succ 9          -- El siguiente de 9
min 28 51       -- El menor entre 28 y 51
succ  2 + 3     -- El sucesor de 2 sumado a 3
succ (2 + 3)    -- El sucesor de 2 + 3
(*) (max 6 4) (succ 9)   -- Notado como prefijo
id 42   -- Identidad
```

Nótese la diferencia entre `f g 2` y `f (g 2)`: En el primer caso llamamos a `f` con **2** argumentos (g y 2), mientras que en el segundo la llamamos con uno sólo (`g 2`).

Haskell incluye una sentencia condicional `if`. **Siempre** tiene que tener una sentencia `else`. Otras estructuras como los bucles no tienen un equivalente en Haskell.

```
if 3 > 2 then 4 else 3
(if False then "Hello" else "Goodbye") ++ "␣World!"
```

FUNCIONES INFIJAS

Haskell permite escribir funciones infijas como prefijas y viceversa:

```
3 + 4      ⟷   (+) 3 4
min 3 2    ⟷   3 `min` 2
```

Sólo necesitamos añadir paréntesis o acentos graves para convertir cualquier función de un tipo al otro.

El sistema de tipos

En Haskell están definidos los tipos básicos comunes a otros lenguajes, y pueden usarse directamente. Podemos comprobar el tipo de cualquier expresión con `:t`. [3] Por ejemplo:

```
True    :: Bool
'a'     :: Char
"a"     :: String
10^320  :: Integer
2       :: Int
3.3     :: Double
```

[3] Algunos de estos ejemplos están simplificados; se describen en la sección de clases de tipos.

Los tipos de Haskell son **estáticos** y **fuertes**. El compilador hace comprobaciones de tipo e infiere los tipos cuando no se declaran explícitamente. La mayoría de errores de un código escrito en Haskell serán de tipos; en otro caso, sólo pueden ser errores de sintaxis o lógicos.

TIPOS FUERTES Y ESTÁTICOS

En un sistema de tipos estáticos tenemos que declarar el tipo de todos los objetos que utilizamos. Es el caso de C++, pero no el de Python o Javascript.

Un sistema de tipos fuertes impide la conversión implícita entre tipos. En C++ los tipos no son fuertes: podemos convertir un entero a un booleano de forma implícita para comprobar si es nulo.

Polimorfismo

Algunos datos pueden tener más de un tipo. En estos casos, Haskell asume el tipo más general e infiere el tipo concreto al usarlo. Podemos indicar el tipo de los objetos que usemos para facilitar la depuración.

```
fst :: (a,b) → a
id  :: a → a
```

Sobre enteros la identidad tendría tipo `id :: Int ->Int`, y sobre booleanos `id :: Bool ->Bool`. Haskell infiere el tipo más general: puede tomar un tipo cualquiera (`a`) y devuelve un elemento de ese tipo. La letra `a` es una **variable de tipo**.

VARIABLES DE TIPO

Las variables de tipo deben empezar con una letra **minúscula**. Usualmente se utiliza una sola letra, pero puede utilizarse cualquier identificador que empiece por minúscula.

En contraposición, los tipos concretos empiezan por una letra **mayúscula** (`Bool` y no `bool`). Si los escribimos en minúscula el compilador pensará que son una variable de tipo.

Clases de tipos

Algunos tipos tienen propiedades comunes. Para aprovecharlas se definen **clases de tipos**, que agrupan tipos con la misma interfaz. La mayoría de los tipos son instancias de `Eq`, porque disponen de una función `(==)` [4]. Las instancias de la clase `Num` pueden sumarse y multiplicarse, las de `Show` pueden convertirse a texto (`String`), y las de `Integral` permiten calcular restos modulares sobre ellas.

[4] Y una función `(/=)` para comprobar la desigualdad.

Las clases de tipos se indican utilizando =>:

```
3    :: Num a ⇒ a
show :: Show a ⇒ a → String
gcd  :: Integral a ⇒ a → a → a
(==) :: Eq a ⇒ a → a → Bool
```

Un tipo puede pertenecer a varias clases: `Integer` pertenece a todas las clases anteriores.

Primeras definiciones

Todo objeto en Haskell es una función: una constante es simplemente una función sin argumentos:

```
constante = 28
```

Definimos las funciones con una serie de igualdades que cubran todas las formas posibles de los argumentos de entrada. El compilador intentará encajar los argumentos en alguna de esas formas en orden y usará la **primera** definición que encaje. Veamos un ejemplo:

```
muestra :: Bool → String
muestra True  = "True"
muestra False = "False"
```

Podemos poner opcionalmente antes de la definición el tipo. Esto se llama **declaración de tipo**. En la mayoría de las ocasiones Haskell es capaz de deducir el tipo de una función:

```
duplica x = x + x
```

Cuando usemos variables estas tienen que empezar por **minúscula** o ser _ [5], que utilizamos para indicar que el argumento no se va a utilizar:

```
esNulo :: Int → Bool
esNulo 0 = True
esNulo _ = False

esPar x = esNulo (mod x 2)
```

Funciones, tipos y currificación

Las funciones en un lenguaje funcional son tratadas como objetos de primera clase. Pueden pasarse como argumentos o ser devueltas como resultado. Por defecto una función en Haskell es currificada.

TIPOS NUMÉRICOS

Los números en Haskell son polimórficos: `3` puede ser interpretado como `Double` o `Int`. Esto permite mantener un sistema de tipos fuerte sin preocuparse por convertirlos explícitamente.

Los tipos más comúnmente utilizados son:

- `Int`: Enteros de 32 o 64 bits.
- `Integer`: Precisión ilimitada.
- `Double`: Punto flotante.

Las clases `Num`, `Integral` y `Fractional` unifican ciertas propiedades de los tipos numéricos. Puedes leer más aquí.

LIBRERÍAS

Cargamos librerías con `import <nombre>`, tanto en un archivo como en `ghci`. Si no están ya instaladas podemos hacerlo utilizando `cabal`.

Para buscar una función concreta existe Hoogle, que permite buscar una función por nombre o por tipo.

[5] Este es el único nombre de variable que puede utilizarse más de una vez

La currificación es la identificación entre las funciones:

$$f : A \times B \to C \qquad f : A \to (B \to C)$$
$$f(a,b) = c \qquad f(a)(b) = c$$

En lugar de tener una función que toma dos argumentos y devuelve un resultado, tenemos una función que toma un argumento y devuelve otra función que toma otro argumento para devolver el resultado. Por ejemplo, la suma queda:

$$\begin{array}{rccc} (+) :: & \mathbb{R} \times \mathbb{R} & \to & \mathbb{R} \\ & x, y & \mapsto & x + y \end{array} \qquad \begin{array}{rccc} (+) :: & \mathbb{R} & \to & (\mathbb{R} \to \mathbb{R}) \\ & x & \mapsto & (y \mapsto x + y) \end{array}$$

De esta forma una función binaria toma un argumento y devuelve una función unaria. Por inducción se definen funciones de más argumentos, toman un sólo argumento y devuelven [6] una función con un argumento menos:

[6] Consideramos $\to$ asociativo por la derecha, de tal forma que `a ->b ->c` es equivalente a `a ->(b ->c)`

```
(&&) :: Bool → Bool → Bool
(+) :: Num a ⇒ a → a → a
const :: a → b → a
(==) :: Eq a ⇒ a → a → Bool
```

Podemos indicar sólo una parte de los argumentos de una función. Esto se conoce como **aplicación parcial**. Aquí definimos la función `xor` y definimos la negación sobre ella, usando primero la definición normal y luego usando aplicación parcial:

```
xor :: Bool → Bool → Bool
xor x y = ((not x) && y) || (x && (not y))

neg' x = xor True x -- Sin aplicación parcial
neg = xor True      -- Con aplicación parcial
```

De la misma forma podemos llamar `(+)` con sólo un argumento, y nos devolverá la función que suma ese número:

```
suma5 = (5+)
suma2 = (2+)
```

Podemos componer funciones con la función composición, `(.)` y aprovechar la aplicación parcial para omitir argumentos.

```
suma7 = suma5 . suma2
```

COMBINANDO FUNCIONES

Existen 3 operadores muy utilizados para combinar funciones:

- `(f . g) x = f (g x)`
- `flip f x y = f y x`
- `f $ x = f x`

`.` es la composición de funciones y `flip` cambia el orden de los argumentos.
`$` es el operador con menor prioridad, lo que permite ahorrar paréntesis:

`succ (9 + 10)` → `succ $ 9 + 10`

Constructores de tipos

Dados los tipos básicos podemos combinarlos para construir nuevos tipos en otras estructuras como listas, árboles o tuplas. Podemos pensar en los constructores de tipos como funciones sobre tipos: Toman uno o más tipos y devuelven un nuevo tipo.

Veamos algunos ejemplos:

Listas

Las listas almacenan colecciones ordenadas de valores de un tipo homogéneo. Su tamaño **no es fijo**, es decir, listas de distintos tamaños pertenecen al mismo tipo.

Una lista se escribe entre corchetes, separando sus elementos por comas. La lista vacía se escribe `[]`:

```
[1,2,3]    :: [Int]
[]         :: [a]
['a','b'] :: [Char]
```

También podemos utilizar el constructor `:`, que antepone un elemento al principio de una lista:

```
[1,2,3] == 1:2:3:[]
```

Algunas funciones relevantes sobre listas son:

- `++ :: [a] ->[a] ->[a]`: concatena dos listas.
- `head :: [a] ->a`: devuelve la cabeza (primer elemento).
- `tail :: [a] ->[a]`: devuelve la cola.
- `null :: [a] ->Bool`: indica si una lista está vacía.
- `elem :: a ->[a] ->Bool`: indica si un elemento está en la lista.
- `(!!) :: [a] ->Int ->a`: elemento en la posición dada.
- `length :: [a] ->Int`: [7] indica la longitud.

Otras funciones generalizan una función binaria sobre una lista. Es el caso de `sum`, `product`, `maximum` o `minimum`.

SINONIMOS DE TIPO

`String` es un sinónimo de tipo de `[Char]`: son distintos nombres para el mismo tipo. La notación usual para cadenas es equivalente a la de listas:

```
"hi`= ['h','i']
```

Los sinónimos de tipo se definen con la palabra clave `type`:

```
type String = [Char]
```

RANGOS

Haskell proporciona una sintaxis de rangos para tipos enumerables, que permite definir listas fácilmente. Prueba estos ejemplos:

- `[1..10]`
- `['a'..'m']`
- `[1,4..20]`
- `[10,8..1]`

Su implementación se basa en la clase de tipos `Enum`.

[7] La mayoría de funciones sobre listas están restringidas sobre `Int` por razones de retrocompatibilidad.

Tuplas

Las tuplas son un tipo que almacena varios datos en una sola estructura. Podemos pensar en ellas como el producto cartesiano de dos tipos.

Podemos guardar por ejemplo una película y su año:

```
(1960, "Psicosis") :: (Int, String)
```

Las tuplas tienen un tamaño *fijo*. Cada n-tupla es un tipo diferente. Pueden contener elementos de distintos tipos.

Es usual utilizar los pares (2-tuplas), como en la función `zip`. Esta función toma dos listas y devuelve una lista de pares con los elementos de cada lista:

```
  zip :: [a] → [b] → [(a,b)]
```

No existen tuplas de un elemento, aunque sí existe la tupla vacía, escrita `()` y llamada *unit*, que funciona como `void`.

Recursividad

La recursividad es la base de Haskell. Como Haskell comprueba en el orden de definición, el caso base debe escribirse antes para que se llegue a ejecutar:

```
factorial 0 = 1
factorial n = n * factorial (n-1)
```

En una función con más de un argumento debemos definir casos base para todos los argumentos[8]:

```
zip :: [a] → [b] → [(a,b)]
zip   []     _     = []
zip   _      []    = []
zip (a:as) (b:bs) = (a,b) : zip as bs
```

[8] En realidad este ejemplo puede hacerse con sólo dos casos (!)

En la mayoría de los casos puede probarse por inducción que la función termina en casos finitos. En las **listas** el caso base suele ser la lista vacía:

```
length :: [a] → Int
length []     = 0
length (_:xs) = 1 + length xs
```

También puede no haber caso base:

```
repeat :: a → [a]
repeat a = a : repeat a
```

Funciones de orden superior

Las funciones de orden superior son aquellas que toman funciones como parámetros. Esto permite modularizar el código. El objetivo es tener funciones simples que se compongan para formar las funciones más complejas:

Tomemos estas dos funciones:

```
mult 1 x = x
mult n x = x + (mult (n-1) x)

replica 1 s = s
replica n s = s ++ (replica (n-1) s)
```

Tienen un patrón común: Aplican una misma función binaria un cierto número de veces. Por tanto podríamos definir:

```
repite :: (a → a → a) → Int → a → a
repite f 1 x = x
repite f n x = f x (repite f (n-1) x)
```

De esta forma: `mult = repite (+)` y `replica = repite (++)`. Esto nos deja unas funciones mucho más limpias y más fáciles de mantener.

Otro ejemplo sería construir una función que aplique otra 2 veces:

```
applyTwice :: (a → a) → a → a
applyTwice f = f . f
```

FUNCIONES LAMBDA

Las funciones lambda son funciones anónimas. Por ejemplo:

```
\x → x + 4
```

Son útiles para ser pasadas como argumento cuando son suficientemente simples. Pueden tomar varios argumentos, que quedan currificados:

```
\x y → x * (y + 2)
```

Funciones de orden superior sobre listas

Las listas son muy importantes en Haskell, y para tratarlas existen varias funciones de orden superior muy utilizadas. Todas estas funciones pueden generalizarse sobre otros tipos por lo que es importante saber utilizarlas:

`map` toma una función y una lista y la devuelve con la función aplicada a sus elementos. Su definición es:

```
map :: (a → b) → [a] → [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

Algunos ejemplos de uso:

```
map (++"!") ["Sugar", "Spice", "Everything␣nice"]
map (map (*3)) [[1,2],[5,2,7],[]]
```

`filter` toma un predicado y una lista, y devuelve una lista con los elementos que cumplen el predicado:

```
filter :: (a → Bool) → [a] → [a]
filter _ [] = []
filter p (x:xs) =
  if p x
    then x : filter p xs
    else filter p xs
```

Un ejemplo es:

```
filter (‘elem‘ ['0'..'9']) "The␣answer␣is␣42"
```

`foldr` [9] toma una función binaria ($\oplus$) y un valor por defecto (z) y aplican la función a los elementos de una lista ordenadamente:

$$\text{foldr } [a_1, a_2, \ldots, a_n] = a_1 \oplus (a_2 \oplus (\cdots \oplus (a_n \oplus z)))$$

[9] Es similar a `reduce` en Python. o `accumulate` en C++. Existe otra función `foldl` que aplica los paréntesis de izquierda a derecha.

Su definición es:

```
foldr :: (a → b → b) → b → [a] → b
foldr f z  []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Estas funciones tienen un gran poder expresivo. Si queremos obtener la suma o el producto de los elementos de una lista podemos hacerlo con `foldr`:

```
sum      =  foldr (+) 0
product  =  foldr (*) 1
concat   =  foldr (++) []
```

Con `foldr` podemos definir `map`, `filter` y la mayoría de funciones sobre listas.

Definición de tipos

Para definir un tipo utilizamos `data`. [10] Por ejemplo, el tipo `Bool` se define:

[10] Existen otras formas de definir tipos. Puedes leer más aquí.

```
  data Bool = False | True
```

Después de `data` indicamos el nombre del tipo y después de `=` indicamos los *constructores de datos*, separados por `|`. Tanto el nombre de tipo como los constructores de datos deben empezar con mayúscula. [11] Podemos pensar en un tipo como `Int` como si estuviera definido:

[11] Excepto si son un símbolo como en los tipos numéricos o en las listas.

```
data Int = -9223372036854775808 | ... | -1 | 0 | 1 | ... | 9223372036854775807
```

Podemos definir tipos con campos que dependan de otros tipos, como por ejemplo puntos en el espacio:

```
data Point = Point Int Int Int
```

Como vemos un tipo puede llamarse igual que uno de sus constructores y para indicar un campo de un cierto tipo ponemos el tipo en el lugar del campo. Los tipos pueden definirse recursivamente, como en el caso de los naturales según los axiomas de Peano:

```
data Nat = Zero | Succ Nat
```

Un natural puede ser el cero o un sucesor de un natural.

También podemos definir **constructores de tipos**. Podemos verlos como funciones sobre tipos: toman como argumento uno o más tipos y definen nuevos tipos. Es el caso del tipo lista o de las tuplas [12]:

```
data [a] = [] | a : [a]
data (a,b) = (a,b)
```

[12] Las definiciones de listas y tuplas son primitivas para poder usar esta notación. Podríamos definir tipos equivalentes utilizando sintaxis legal.

Después del nombre del constructor de tipos damos un nombre a las variables que utiliza (`a`). Podemos ver entonces que `[a]` puede ser una lista vacía (`[]`) o un par de elementos, el primero de tipo `a` y el segundo una lista de tipo `a`.

Los constructores de datos se utilizan para definir funciones en lo que se conoce como *reconocimiento de patrones*. Cuando definimos una función podemos emplear una variable para un argumento o bien un constructor de datos. Por ejemplo[13]:

```
fst :: (a,b) → a
fst (a,b) = a
```

[13] Podemos usar el mismo nombre para las variables de tipo y los nombres de los elementos del constructor porque sus ámbitos son distintos.

Como vimos antes en el caso de las listas es habitual utilizar el constructor `:` [14]:

```
head (x:xs) = x
second (x:y:xs) = y
```

[14] Debemos poner `head (x:xs)` y no `head x:xs` ya que si no nuestra función estaría tomando 3 argumentos: `x`, `:` y `xs`

Otro ejemplo es `(++)`:

```
(++) :: [a] → [a] → [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Podemos utilizar el reconocimiento de patrones sobre cualquier tipo que soporte la igualdad utilizando cualquier constructor del tipo. Debemos asegurarnos de que cubrimos todos los constructores.

Apéndice

En este apéndice se incluyen otros temas de interés sobre la sintaxis o uso de Haskell.

Comprensión de listas

La comprensión de listas es una forma de escribir listas parecida a la notación para expresar conjuntos por extensión. Su sintaxis es:

```
-- [<expresion> | <variable> <- <lista>, <predicado>]
[x*2 | x <- [1..10]] -- [2,4,6,8,10,12,14,16,18,20]
[x*2 | x <- [1..10], x*2 >= 12] -- [12,14,16,18,20]
```

Cada elemento de la lista es sustituido en la expresión, que puede depender de la variable, y, si cumple los predicados se añadirá a la lista. Si se utilizan varios predicados éstos se separan por comas. También podemos utilizar varias listas, cada una ligada a una variable. En este caso las variables se sustituirán por cada combinación posible de los elementos:

```
[ (x,y) | x <- [1,2,3], y <- [8,3], x < y]
-- [(1,8),(1,3),(2,8),(2,3),(3,8)]
```

Listas infinitas y evaluación perezosa

La sintaxis de rangos nos permite escribir listas infinitas:

```
[1..] -- Enteros positivos
fibs = 0 : 1 : zipWith (+) fibs (tail fibs) --
    Secuencia de Fibonacci
repeat :: a → [a] -- Crea listas infinitas con todos
    los elementos iguales
repeat n = n : repeat n -- repeat n = [n,n..]
```

Si intentamos mostrar estas listas en el intérprete no acabará. Sin embargo, podemos utilizarlas junto con otras funciones sin problema:

```
  zip [1..] ["azul","verde","rojo"] -- [(1,"azul"),(2,"
      verde"),(3,"rojo")]
```

Esto sucede porque Haskell es un lenguaje de evaluación no estricta o perezoso, por lo que sólo evaluará los elementos que necesite. Así, en `||`, el segundo elemento sólo se evaluará si el primero es `False`, ya que si es `True` el valor de la expresión ya estará fijado.

Prueba del compilador de Haskell

Escribimos el clásico programa *"Hello World"* para Haskell, que servirá para probar el compilador. En `test.hs` escribimos el programa:

```
main = putStrLn "Hello World!"
```

Compilamos usando `ghc -o hello test.hs` [15]. Acabamos de definir la función `main` como la función `putStrLn` aplicada sobre la cadena "`Hello World!`".

[15] También podríamos usar `runghc`, que compila y ejecuta el programa en un solo paso

A diferencia de lo que haríamos en un lenguaje imperativo no le decimos que para ejecutar `main` haya que ejecutar una función `putStr`, sino que **definimos** `main` como la función.

Además del compilador, *Haskell Platform* incluye `runhaskell`, un intérprete no interactivo que podríamos llamar con `runhaskell test.hs` para este ejemplo.

Ejemplos prácticos y más

En la carpeta `examples` hay algunos ejemplos prácticos de programas completos. Si quieres seguir aprendiendo, recomendamos los siguientes recursos:

- Getting started with Haskell
- Try Haskell
- ¡Aprende Haskell por el bien de todos!
- Wikibooks - Haskell
- Learning Haskell
- Real World Haskell