Introducción a Haskell

Pablo F. Castro

Programación Avanzada, Universidad Nacional de Río Cuarto, Departamento de Computación

2015

Introducción a Haskell

Haskell fue introducido en 1987 con el objetivo de introducir un lenguaje funcional moderno. Existen diversos interpretes y compiladores:

- Hugs, es un interprete de Haskell muy usado, se puede obtener en www.haskell.org/hugs
- Glasgow Haskell, es un compilador para Haskell se puede obtener en http://www.haskell.org/ghc/.

En la materia utilizaremos **Hugs**.

Tipos Básicos de Haskell: Bool

El tipo `Bool` contiene dos valores: `True, False` y las siguientes operaciones:

- `Not :: Bool -> Bool`, negación lógica.
- `&& :: Bool -> Bool -> Bool`, conjunción lógica.
- `|| :: Bool -> Bool -> Bool`, disyunción lógica.

Estos tipos están definidos por pattern matching a la izquierda. Por ejemplo, la expresión:

```
False && (infty == 3)
```

se evalua a `False`, pero:

```
(infty == 3) && False
```

Es indefinido.

Tipos Básicos de Haskell:Char, Int, Integer, Floats

El tipo Char contiene los valores `'a'`,`'b'`,`'c'`,`...`, y las siguientes funciones:

- `ord::Char -> Int` convierte carácteres a enteros.
- `chr:Int -> Char` convierte enteros a carácteres.

También tenemos los tipos:

- númericos `Int` (enteros de precisión fija),
- `Integer` (enteros de precisión variable),
- `Float` (números reales).

Tuplas

Usando los tipos básicos podemos construir tuplas y listas. Dados `A` y `B`:

`(A,B)`

Es el tipo de tuplas de `A` y `B`. Por ejemplo:

- `(True,False) :: (Bool, Bool)`
- `(True, 1) :: (Bool, Int)`
- `([],[]) :: ([A],[B])`

Tenemos dos operaciones:

- `fst :: (A,B) -> A` y
- `snd :: (A,B) -> B`.

Funciones

Dados dos tipos `A` y `B`, `A->B` es el tipo de las funciones de `A` en `B`. Por ejemplo:

```
Not :: Bool -> Bool
```

Las funciones de **Alto Orden** son aquellas que toman como parámetros funciones, por ejemplo:

```
. :: (b -> c) -> (a -> b) -> (a -> c)
```

Ejercicio: Definir la composición de funciones.

Listas

El tipo `[A]` contiene todas las listas de tipo `A`, en donde:

- `[]` es la lista vacía.
- `x:xs` es la lista que tiene como primer elemento *x* y *xs* es el resto.

Todos los elementos de la lista son del mismo tipo. Podemos considerar también listas infinitas, por ejemplo:

```
ones :: [Int]
ones = 1 : ones
```

Tenemos un conjunto importante de funciones sobre listas:

- `head :: [A] -> A`, devuelve la cabeza de la lista.
- `last :: [A] -> A`, devuelve el último elemento.
- `tail :: [A] -> [A]`, devuelve la cola de la lista.

Definición por Casos

En haskell podemos utilizar definición por casos:

```
sign :: Int -> Int
sign x | x >= 0 = 1
       | x < 0 =-1
```

También podemos usar pattern matching:

```
take 0     _      = []
take _     []     = []
take n     (x:xs) = x : take (n-1) xs
```

Operaciones sobre Listas

- `init :: [A] -> [A]`, devuelve el comienzo de una lista.
- `length :: [A] -> Int`, devuelve la longitud de una lista.
- `!! :: [A] -> A -> [A]`, devuelve el elemento en la i-ésima posición.

Funciones de alto orden sobre listas:

```
foldl :: (a->b->a) -> a -> [b] ->a
foldl f z [] = z
foldl f z x:xs = foldl f (f z x) xs
```

Es decir:

$$foldl \oplus z\ (x_1 : (x_2 : (x_3 : (x_4 : ... : (x_n : [])))))$$
$$=$$
$$(((z \oplus x_1) \oplus x_2) \oplus x_3) \oplus x_4 \ldots$$

Operaciones sobre listas

Por ejemplo, la siguiente expresión: `foldl (+) 0 xs` calcula `sum xs`. De la misma forma tenemos:

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

Es decir:

$$foldr \oplus z\ x_1 : (x_2 : (x_3 : (... : (x_n : []))))$$
$$=$$
$$x_1 \oplus (x_2 \oplus \ldots (x_n \oplus z) \ldots)$$

Operaciones sobre listas

La función `Map:(A -> B) -> [A] -> [B]`, está definida como:

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : (map f xs)
```

Es decir:

$$map\ f\ [x_0, x_1, x_2, x_3, x_4, \ldots] = [f(x_0), f(x_1), f(x_2), f(x_3), f(x_4), \ldots]$$

Ejemplo:

```
square x = x*x

squarel :: [Int] -> [Int]
squarel xs = map square xs
```

También podríamos haber dicho: **squarel = map square**

Operaciones sobre Listas

Las siguientes operaciones son útiles:

- *zip* :: $[a] \to [b] \to [(a, b)]$, pone los elementos de la primera lista y la segunda en pares.
- *zipWith* :: $(a \to b \to c) \to [a] \to [b] \to [c]$, combina los elementos de las listas usando *f*.
- *filter* :: $(a -> Bool) -> [a] -> [a]$, filtra los elementos de la lista usando *p*.
- *takeWhile* :: $(a -> Bool) -> [a] -> [a]$, se queda con los primero elementos que cumplen con *p*.
- *dropWhile* :: $(a -> Bool) -> [a] -> [a]$, tira los primeros elementos que cumplen con *p*.

Ejemplos:

- foldl 0 max (el máximo de una lista)
- filter ((0 /=) . (‘mod‘ 2)) (los impares de una lista)
- zipWith (+) xs ys (suma dos listas elemento por elemento)
- ns = 0 : map (+1) ns (devuelve los naturales)

Comprensión sobre Listas

En conjuntos podemos hacer:

$$\{2 * x \mid x \in \{0, 1, 2, 3, 4\}\}$$

En Haskell tenemos la comprensión de listas, por ejemplo:

```
[2 * x | x <- [0,1,2,3,4]]
```

La expresión $x < -[0, 1, 2, 3, 4]$ es llamada **generador**.

Podemos tener mas de un generador:

```
[(x,y) | x <- [0,1], y <- [4,5]]
```

que devuelve: $[(0, 4), (0, 5), (1, 4), (1, 5)]$

Listas por Comprensión

Podemos usar guardas para producir listas más interesantes:

```
[x | x <- [0..], even x]
```

Podemos usar comprensión de listas para definir funciones:

```
divisores :: Int -> [Int]
divisores n = [x | x <- [1..n], n mod x = 0]
```

Ejercicio: Definir la función `prime` usando `divisores`.

Ejercicios con Comprensión de Listas

- calcular $1^2 + 2^2 + \cdots + 100^2$ usando comprensión de listas.
- Tres números x, y, z se dicen Pitagóricos is $x^2 + y^2 = z^2$, usando comprensión calcular las lista de todos las triplas Pitagóricas hasta un n dado.
- Calcular la lista de números perfectos utilizando comprensión de listas.

Declarando Nuevos Tipos

Podemos definir nuevos tipos con el constructor `type`, por ejemplo:

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

Podemos definir un tablero como:

```
type Board = [Pos]
```

en donde:

```
type Pos = (Int, Int)
```

También podemos definir tipos completamente nuevos con `Data`.

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

Tipos Recursivos

Podemos introducir tipos recursivos (naturales, listas, árboles, etc). Por ejemplo:

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

Es decir, los naturales son:

```
Zero, Succ Zero, Succ(Succ Zero), Succ(Succ(Succ Zero)), ...
```

Podemos definir listas:

```
data List a = Nil | Cons a (List a)
```

Árboles binarios:

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

Ejercicios

- Definir las siguientes funciones sobre árboles:
  - La función `size:Tree a -> Int`, que dado un árbol devuelve el número de nodos del árbol.
  - La función `alt: Tree a -> Int`, que me da la altura de un árbol.
  - La función `bal: Tree a -> Bool`, que dice si un árbol es balanceado.
  - La función `flatten: Tree a -> [a]` que transforma un árbol en una lista.
- Probar por inducción que: $alt\ t \leq size\ t$
- Definir la función *map* para árboles.

Clases

Sobrecarga de Operadores

Un operador se dice sobrecargado si puede utilizarse con diferentes tipos.

Las clases nos permiten definir funciones sobre tipos que comparten las mismas operaciones:

```
elem x [] = False
elem x (y:ys) | x == y = True
              | otherwise = elem x ys
```

Solo funciona sobre tipos que tienen la igualdad definida, esto se escribe en Haskell:

```
elem :: (Eq a) => a -> [a] -> Bool
```

La clase *Eq* esta conformada por todos los tipos que tienen la igualdad:

```
Class Eq a where (==):: a -> a -> Bool
```

Instanciando la igualdad para Tree

Podemos decir que *Nat* pertenece a *Eq* declarando:

```
instance (Eq a) => Eq Nat where
Zero == Zero = True
Zero == Succ n = False
Succ n == Succ m = n == m
```

Ejercicio: Instanciar *Eq* para árboles.

Otras clases:

- *Show*, aquellos tipos que se pueden mostrar por pantalla.
- *Ord*, tipos con una relación de orden.

Ejemplo Final:QuickSort

podemos programar el QuickSort de la siguiente forma:

```
qsort [] = []
qsort (x:xs) = qsort left ++ [x] ++ qsort right
          where
          left = [a | a <- xs, a <= x ]
          right = [b | b <- xs, b > x ]
```