

Programación Funcional con Haskell

Mauro Jaskelioff

28/03/2017



Hay dos maneras de construir un diseño de software:

Una manera es hacerlo de manera tan simple que sea obvio que no hay deficiencias.

La otra es hacerlo de manera tan complicada que no haya deficiencias obvias.

Tony Hoare, 1980

Programación funcional

- ▶ La programación funcional es un estilo de programación moderno.
- ▶ No usa un modelo de computación basado en máquinas sino en un lenguaje simple y elegante (el λ -cálculo).
- ▶ Su simpleza y versatilidad lo hacen ideal para aprender conceptos de
 - ▶ programación,
 - ▶ lenguajes de programación,
 - ▶ computabilidad,
 - ▶ semántica,
 - ▶ verificación, derivación, testing.

¿Qué es la programación funcional?

Hay diferentes opiniones.

- ▶ Es un *estilo* de programación en el cual el método básico de computar es aplicar funciones a argumentos
- ▶ Un lenguaje de programación funcional es un lenguaje que *permite y alienta* el uso de un estilo funcional.

- ▶ Haskell posee varias ventajas.
 - ▶ Programas Concisos
 - ▶ Sistemas de Tipos Poderoso
 - ▶ Funciones Recursivas
 - ▶ Fácilidad para probar propiedades de programas.
 - ▶ Funciones de Alto Orden
 - ▶ Evaluación Perezosa
 - ▶ Facilidad para definir DSLs
 - ▶ Efectos Monádicos
- ▶ Haremos un repaso por la sintaxis básica de Haskell.
- ▶ También introduciremos algunos conceptos nuevos.

Preludio

- ▶ Muchas funciones de uso común están definidas en el Preludio (Prelude.hs)
- ▶ Además de las operaciones aritméticas usuales se definen muchas funciones que operan sobre *listas*.
 - ▶ *head, tail, (!!), take, drop, length, sum, product, (++)*, *reverse*
- ▶ Leer el código del preludio puede ser muy instructivo (y sorprendente!).

- ▶ Usaremos GHC, un compilador (*ghc*) e intérprete (*ghci*) de Haskell, que también soporta varias extensiones del mismo.
- ▶ URL: <http://www.haskell.org/ghc>
- ▶ Puede ser conveniente instalar la *Haskell Platform*, que consiste del GHC junto con una colección de bibliotecas frecuentemente usadas.

La aplicación de funciones

- ▶ en Haskell la aplicación se denota con un espacio y asocia a la izquierda.

Matemáticas	Haskell
$f(x)$	$f\ x$
$f(x, y)$	$f\ x\ y$
$f(g(x))$	$f\ (g\ x)$
$f(x, (g(y)))$	$f\ x\ (g\ y)$
$f(x)\ g(y)$	$f\ x\ * \ g\ y$

- ▶ La aplicación tiene mayor precedencia que cualquier otro operador.

$$f\ x + y = (f\ x) + y$$

Nombres y comentarios

- ▶ Las funciones y sus argumentos deben empezar con minúscula, y pueden ser seguidos por cero o más letras (mayúsculas o minúsculas), dígitos, guiones bajos, y apóstrofes.
- ▶ Las palabras reservadas son:

**case class data default deriving do else if import in infix
infixl infixr instance let module newtype of then type where**

- ▶ Los comentarios se escriben

-- comentario que finaliza junto con la línea

{- bloque de comentarios, útil para comentarios
que no entran en una sola línea. -}

Offside Rule

- ▶ En una serie de definiciones, cada definición debe empezar en la misma columna.

$$a = b + c$$

where

$$b = 1$$

$$c = 2$$

$$d = a + 2$$

- ▶ Gracias a esta regla no hace falta un sintaxis *explícita* para agrupar definiciones.

Operadores infijos

- ▶ Los operadores infijos son funciones como cualquier otra.
- ▶ Una función se puede hacer infija con *backquotes*

$$10 \text{ 'div' } 4 = \text{div } 10 \ 4$$

- ▶ Se pueden definir operadores infijos usando alguno de los símbolos disponibles

$$a ** b = (a * b) + (a + 1) * (b - 1)$$

- ▶ La asociatividad y precedencia se indica usando **infixr** (asociativad der.), **infixl** (asociatividad izq.), o **infix** (si los paréntesis deben ser obligatorios)

$$\text{infixr } 6 \ (**)$$

- ▶ Un tipo es un nombre para una colección de valores
 - ▶ Ej: *Bool* contiene los valores *True* y *False*.
 - ▶ Escribimos *True :: Bool* y *False :: Bool*.
- ▶ En general, si una expresión *e* tiene tipo *t* escribimos

$$e :: t$$

- ▶ **En Haskell, toda expresión válida tiene un tipo**
- ▶ El tipo de cada expresión es calculado previo a su evaluación mediante la *inferencia de tipos*.
- ▶ Si no es posible encontrar un tipo (por ejemplo (*True* + 4)) el compilador/intérprete protestará con un *error de tipo*.

Tipos básicos

Alguno de los tipos básicos de Haskell son:

- ▶ *Bool*, booleanos
- ▶ *Char*, caracteres
- ▶ *Int*, enteros de precisión fija.
- ▶ *Integer*, enteros de precisión arbitraria.
- ▶ *Float*, números de punto flotante de precisión simple.

- ▶ Una lista es una secuencia de valores del *mismo* tipo
 - ▶ $[True, True, False, True] :: [Bool]$
 - ▶ $['h', 'o', 'l', 'a'] :: [Char]$
- ▶ En general, $[t]$ es una lista con elementos de tipo t .
- ▶ t , puede ser *cualquier* tipo válido.
 - ▶ $[['a'], ['b', 'c'], []] :: [[Char]]$
- ▶ No hay restricción con respecto a la longitud de las listas.

Tuplas

- ▶ Una tupla es una secuencia finita de valores de tipos (posiblemente) distintos.
 - ▶ $(True, True) :: (Bool, Bool)$
 - ▶ $(True, 'a', 'b') :: (Bool, Char, Char)$
- ▶ En general, (t_1, t_2, \dots, t_n) es el tipo de una n -tupla cuyas componente i tiene tipo t_i , para i en $1 \dots n$.
- ▶ A diferencia de las listas, las tuplas tienen explicitado en su tipo la *cantidad* de elementos que almacenan.
- ▶ Los tipos de las tuplas no tiene restricciones
 - ▶ $('a', (True, 'c')) :: (Char, (Bool, Char))$
 - ▶ $((['a', 'b'], 'a'), 'b') :: (([Char], Char), Char)$

- ▶ Una función mapea valores de un tipo en valores de otro
 - ▶ $not :: Bool \rightarrow Bool$
 - ▶ $isDigit :: Char \rightarrow Bool$
- ▶ En general, Un tipo función $t_1 \rightarrow t_2$ mapea valores de t_1 en valores de t_2 .
- ▶ Se pueden escribir funciones con múltiples argumentos o resultados usando tuplas y listas.

$add \quad \quad \quad :: (Int, Int) \rightarrow Int$

$add(x, y) = x + y$

$deceroa \quad :: Int \rightarrow [Int]$

$deceroa\ n = [0..n]$

Curricación y aplicación parcial

- ▶ Otra manera de tomar múltiples argumentos es devolver una función como resultado

$$add' :: Int \rightarrow (Int \rightarrow Int)$$
$$add' \ x \ y = x + y$$

- ▶ A diferencia de add , add' toma los argumentos de a uno por vez. Se dice que add' está curricada.
- ▶ La ventaja de la versión curricada es que permite la *aplicación parcial*:

$$suma3 :: Int \rightarrow Int$$
$$suma3 = add' \ 3$$

- ▶ Si queremos expresar una función que tome 3 argumentos devolvemos una función que devuelve una función:

$$\begin{aligned}mult &:: Int \rightarrow (Int \rightarrow (Int \rightarrow Int)) \\mult\ x\ y\ z &= x * y * z\end{aligned}$$

- ▶ Para evitar escribir muchos paréntesis, por convención el constructor de tipos \rightarrow asocia a la derecha.
 $mult :: Int \rightarrow Int \rightarrow Int \rightarrow Int$
- ▶ Notar que esta convención es consistente con la aplicación asociando a la izquierda.
- ▶ En Haskell todas las funciones están curricadas (salvo algunos casos particulares).

Nombres de los tipos

- ▶ A excepción de listas, tuplas y funciones, los nombres de los tipos concretos comienzan con mayúsculas.
- ▶ El espacio de nombres de los tipos está completamente separado del espacio de nombres de las expresiones.

- ▶ Una función es polimórfica si su tipo contiene variables de tipo.

$length :: [a] \rightarrow Int$

Para cualquier tipo a la función $length$ es la misma.

- ▶ Las variables de tipo se escriben con minúscula.
- ▶ Las variables de tipo pueden ser *instanciadas* a otros tipos

$length [False, True] \quad \leftarrow a = Bool$

$length ['a', 'b'] \quad \leftarrow a = Char$

- ▶ A veces se llama polimorfismo paramétrico a este tipo de polimorfismo.

Sobrecarga de funciones

- ▶ ¿Cuál es el tipo de $3 + 2$?
- ▶ En Haskell, los números y las operaciones aritméticas están sobrecargadas
- ▶ Esta sobrecarga se realiza mediante *clases de tipo*.
- ▶ El operador $(+)$ tiene tipo:

$$(+) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$$

- ▶ $(+)$ está definido para cualquier tipo que sea una *instancia* de la clase *Num*.
- ▶ A diferencia del polimorfismo paramétrico, hay una definición distinta de $(+)$ para cada instancia.

Clases de tipo

- ▶ Haskell provee una serie de clases básicas:
- ▶ *Eq* son los tipos cuyos valores pueden ser comparados para ver si son iguales o no.

$$(==) :: Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool$$
$$(/=) :: Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool$$

¿Qué tipo no puede ser instancia de esta clase?

- ▶ *Ord* son los tipos que además de ser instancias de *Eq* poseen un orden total.

$$(<), (\leq), (>), (\geq) :: Ord\ a \Rightarrow a \rightarrow a \rightarrow Bool$$
$$min, max :: Ord\ a \Rightarrow a \rightarrow a \rightarrow a$$

Clases de tipo

- ▶ *Show* son los tipos cuyos valores pueden ser convertidos en una cadena de caracteres.

$$show :: Show\ a \Rightarrow a \rightarrow String$$

- ▶ *Read* es la clase dual. Son los tipos que se pueden obtener de una cadena de caracteres.

$$read :: Read\ a \Rightarrow String \rightarrow a$$

- ▶ ¿A qué tipo corresponde la instancia que leerá
 - ▶ *not* (*read* "False")?
 - ▶ *read* "3"?
 - ▶ *read* "3" :: *Int*?

Clases de Tipo

- ▶ *Num* son los tipos numéricos
- ▶ Sus instancias deben implementar

$(+), (-), (*) \quad :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$
 $\text{negate}, \text{abs}, \text{signum} :: \text{Num } a \Rightarrow a \rightarrow a$

¿Y la división?

- ▶ *Integral* son los tipos que son *Num* e implementan

$\text{div}, \text{mod} :: \text{Integral } a \Rightarrow a \rightarrow a \rightarrow a$

- ▶ *Fractional* son los tipos que son *Num* e implementan

$(/) \quad :: \text{Fractional } a \Rightarrow a \rightarrow a \rightarrow a$
 $\text{recip} :: \text{Fractional } a \Rightarrow a \rightarrow a$

Expresiones Condicionales

- ▶ Las funciones pueden ser definidas usando expresiones condicionales

$$\begin{aligned} \text{abs} &:: \text{Int} \rightarrow \text{Int} \\ \text{abs } n &= \text{if } n \geq 0 \text{ then } n \text{ else } -n \end{aligned}$$

- ▶ Para que la expresión condicional tenga sentido, ambas ramas de la misma deben tener el mismo tipo.
- ▶ Las expresiones condicionales siempre deben tener la rama “else”
- ▶ Por lo tanto no hay ambigüedades en caso de anidamiento:

$$\begin{aligned} \text{signum} &:: \text{Int} \rightarrow \text{Int} \\ \text{signum } n &= \text{if } n < 0 \text{ then } -1 \text{ else} \\ &\quad \text{if } n == 0 \text{ then } 0 \text{ else } 1 \end{aligned}$$

Ecuaciones con Guardas

- ▶ Una alternativa a los condicionales es el uso de ecuaciones con guardas.

$$\begin{array}{lcl} \text{abs } n & | & n \geq 0 \\ & | & \text{otherwise} \end{array} \begin{array}{l} = n \\ = -n \end{array}$$

- ▶ Se usan para hacer ciertas definiciones más fáciles de leer.

$$\begin{array}{lcl} \text{signum } n & | & n < 0 \\ & | & n == 0 \\ & | & \text{otherwise} \end{array} \begin{array}{l} = -1 \\ = 0 \\ = 1 \end{array}$$

- ▶ La condición *otherwise* se define en el preludio como

$$\text{otherwise} = \text{True}$$

Pattern Matching

- ▶ Muchas funciones se definen más claramente usando *pattern matching*.

$$\begin{aligned} \text{not} &:: \text{Bool} \rightarrow \text{Bool} \\ \text{not False} &= \text{True} \\ \text{not True} &= \text{False} \end{aligned}$$

- ▶ Los patrones se componen de constructores de datos y variables (salvo los patrones 0 y $n + 1$) .
- ▶ Una variable es un patrón que nunca falla.

$$\begin{aligned} \text{succ} &:: \text{Int} \rightarrow \text{Int} \\ \text{succ } n &= n + 1 \end{aligned}$$

Pattern Matching

- Usando el ingenio se pueden obtener definiciones concisas.

$$\begin{aligned}(\wedge) & \quad \quad \quad :: \textit{Bool} \rightarrow \textit{Bool} \rightarrow \textit{Bool} \\ \textit{True} \wedge \textit{True} &= \textit{True} \\ \textit{True} \wedge \textit{False} &= \textit{False} \\ \textit{False} \wedge \textit{True} &= \textit{False} \\ \textit{False} \wedge \textit{False} &= \textit{False}\end{aligned}$$

puede ser escrita en forma compacta como

$$\begin{aligned}(\wedge) & \quad \quad \quad :: \textit{Bool} \rightarrow \textit{Bool} \rightarrow \textit{Bool} \\ \textit{True} \wedge \textit{True} &= \textit{True} \\ _ \wedge _ &= \textit{False}\end{aligned}$$

- Notar la importancia del orden de las ecuaciones.

Patrones de tuplas

- ▶ Una tupla de patrones es un patrón.

$$\text{fst} \quad \quad \quad :: (a, b) \rightarrow a$$

$$\text{fst } (x, -) = x$$

$$\text{snd} \quad \quad \quad :: (a, b) \rightarrow b$$

$$\text{snd } (-, y) = y$$

- ▶ ¿Qué hace la siguiente función?

$$f \ (x, (y, z)) = ((x, y), z)$$

- ▶ En general, los patrones pueden anidarse

Patrones de Listas

- ▶ Toda lista (no vacía) se contruye usando el operador $(:)$ (llamado *cons*) que agrega un elemento al principio de la lista

$$[1, 2, 3, 4] \mapsto 1 : (2 : (3 : (4 : [])))$$

- ▶ Por lo tanto, puedo definir funciones usando el patrón $(x : xs)$

head $:: [a] \rightarrow a$

head $(x : _) = x$

tail $:: [a] \rightarrow [a]$

tail $(_ : xs) = xs$

- ▶ $(x : xs)$ sólo matchea el caso de listas no vacías $> head []$
Error!

Expresiones Lambda

- ▶ Se pueden construir funciones sin darles nombres usando *expresiones lambda*.

$$\lambda x \rightarrow x + x$$

- ▶ Estas funciones se llaman *funciones anónimas*.
- ▶ En Haskell escribimos `\` en lugar de la letra griega lambda λ
- ▶ La notación proviene del cálculo lambda, en el cual se basa Haskell, y que estudiaremos en detalle más adelante.

Utilidad de las expresiones lambda

- ▶ Usando lambdas puedo explicitar que las funciones son simplemente valores:

$$add\ x\ y = x + y$$

$$add = \lambda x \rightarrow (\lambda y \rightarrow x + y)$$

- ▶ Evito tener que darle un nombre a una función que se usa una sola vez.

$$\begin{aligned} impares\ n &= map\ f\ [0..n-1] \\ &\quad \textbf{where}\ f\ x = x * 2 + 1 \end{aligned}$$

$$odds\ n = map\ (\lambda x \rightarrow x * 2 + 1)\ [0..n-1]$$

- Un operador infijo, puede ser escrito en forma prefija usando paréntesis:

$$\begin{array}{c} > (+) 1 2 \\ 3 \end{array}$$

- También uno de los argumentos puede ser incluido en los paréntesis

$$\begin{array}{c} > (1+) 2 \\ 3 \\ > (+2) 1 \\ 3 \end{array}$$

- En general, dado un operador \oplus , entonces las funciones de la forma (\oplus) , $(x\oplus)$, $(\oplus y)$ son llamadas *secciones*.

Conjuntos por comprensión

- En matemáticas, una manera de construir conjuntos a partir de conjuntos existentes es con la notación por comprensión

$$\{x^2 | x \in \{1 \dots 5\}\}$$

describe el conjunto $\{1, 4, 9, 16, 25\}$ o (lo que es lo mismo) el conjunto de todos los números x^2 tal que x sea un elemento del conjunto $\{1 \dots 5\}$

Listas por comprensión

- En Haskell, una manera de construir listas a partir de listas existentes es con la notación por comprensión

$$[x \uparrow 2 \mid x \leftarrow [1..5]]$$

describe la lista $[1, 4, 9, 16, 25]$ o (lo que es lo mismo) la lista de todos los números $x \uparrow 2$ tal que x sea un elemento de la lista $[1..5]$

- ▶ La expresión $x \leftarrow [1..5]$ es un *generador*, ya que dice como se generan los valores de x .
- ▶ Una lista por comprensión puede tener varios generadores, separados por coma.

> $[(x, y) \mid x \leftarrow [1, 2, 3], y \leftarrow [4, 5]]$
 $[(1, 4), (1, 5), (2, 4), (2, 5), (3, 4), (3, 5)]$

- ▶ ¿Qué pasa cuando cambiamos el orden de los generadores?

> $[(x, y) \mid y \leftarrow [4, 5], x \leftarrow [1, 2, 3]]$

- ▶ Los generadores posteriores cambian más rápidamente.

Generadores dependientes

- Un generador puede depender de un generador anterior

$$[(x, y) \mid x \leftarrow [1..3], y \leftarrow [x..3]]$$

Esto es la lista de todos los pares (x, y) tal que x, y están en $[1..3]$ e $y \geq x$.

- ¿Qué hace la siguiente función?

concat $:: [[a]] \rightarrow [a]$
concat $xss = [x \mid xs \leftarrow xss, x \leftarrow xs]$

$> \text{concat } [[1, 2, 3], [4, 5], [6]]$
 $[1, 2, 3, 4, 5, 6]$

Guardas

- ▶ Las listas por comprensión pueden usar guardas para restringir los valores producidos por generadores anteriores

$$[x \mid x \leftarrow [1..10], \text{even } x]$$

- ▶ ¿Qué hace la siguiente función?

$$\text{factors} :: \text{Int} \rightarrow [\text{Int}]$$
$$\text{factors } n = [x \mid x \leftarrow [1..n], n \text{ 'mod' } x == 0]$$

- ▶ Como un número n es primo iff sus únicos factores son 1 y n , podemos definir

$$\text{prime} :: \text{Int} \rightarrow \text{Bool}$$
$$\text{prime } n = \text{factors } n == [1, n]$$
$$\text{primes} :: \text{Int} \rightarrow [\text{Int}]$$
$$\text{primes } n = [x \mid x \leftarrow [2..n], \text{prime } x]$$

- ▶ Una *String* es una lista de caracteres.
- ▶ "Hola" :: *String*
- ▶ "Hola" = ['H', 'o', 'l', 'a']
- ▶ Todas las funciones sobre listas son aplicables a *String*, y las listas por comprensión pueden ser aplicadas a *Strings*.

cantminusc :: *String* → *Int*

cantminusc xs = length [x | x ← xs, isLower x]

- La función *zip*, mapea dos listas a una lista con los pares de elementos correspondientes

$$\begin{aligned} \text{zip} &:: [a] \rightarrow [b] \rightarrow [(a, b)] \\ &> \text{zip } ['a', 'b', 'c'] [1, 2, 3, 4] \\ &[('a', 1), ('b', 2), ('c', 3)] \end{aligned}$$

- Ejemplo: Lista de pares de elementos adyacentes:

$$\begin{aligned} \text{pairs} &:: [a] \rightarrow [(a, a)] \\ \text{pairs } xs &= \text{zip } xs \text{ (tail } xs) \end{aligned}$$

- ¿Está una lista ordenada?

$$\begin{aligned} \text{sorted} &:: \text{Ord } a \Rightarrow [a] \rightarrow \text{Bool} \\ \text{sorted } xs &= \text{and } [x \leq y \mid (x, y) \leftarrow \text{pairs } xs] \end{aligned}$$

Ejemplo *zip*: pares índice/valor

- Podemos usar *zip* para generar índices

$$\begin{aligned} \text{rangeof} & \quad :: \text{Int} \rightarrow \text{Int} \rightarrow [a] \rightarrow [a] \\ \text{rangeof } \text{low } \text{hi } \text{xs} &= [x \mid (x, i) \leftarrow \text{zip } \text{xs } [0..], \\ & \quad i \geq \text{low}, \\ & \quad i \leq \text{hi}] \end{aligned}$$
$$\begin{aligned} &> [x \uparrow 2 \mid x \leftarrow [1..10]] \\ &[1, 4, 9, 16, 25, 36, 49, 64, 81, 100] \\ &> \text{rangeof } 3 \ 7 [x \uparrow 2 \mid x \leftarrow [1..10]] \\ &[16, 25, 36, 49, 64] \end{aligned}$$

Recursión

- ▶ En los lenguajes funcionales, la recursión es uno de los principales recursos para definir funciones.

factorial $:: \text{Int} \rightarrow \text{Int}$

factorial 0 = 1

factorial $n = n * \text{factorial } (n - 1)$

- ▶ ¿Qué sucede con *factorial* n cuando $n < 0$?
- ▶ Recursión sobre listas

length $:: [a] \rightarrow \text{Int}$

length [] = 0

length $(x : xs) = 1 + \text{length } xs$

Recursión Mutua

- ▶ No hace falta ninguna sintaxis especial para la recursión mutua.

$$\text{zigzag} :: [(a, a)] \rightarrow [a]$$
$$\text{zigzag} = \text{zig}$$
$$\text{zig} [] = []$$
$$\text{zig} ((x, -) : xs) = x : \text{zag} xs$$
$$\text{zag} [] = []$$
$$\text{zag} ((-, y) : xs) = y : \text{zig} xs$$

- ▶ $\text{zigzag} [(1, 2), (3, 4), (5, 6), (7, 8)]$
 $[1, 4, 5, 8]$

Quicksort

- ▶ El algoritmo de ordenación Quicksort:
 - ▶ La lista vacía está ordenada
 - ▶ Las listas no vacías pueden ser ordenadas, ordenando los valores de la cola \leq que la cabeza, ordenando los valores $>$ que la cabeza y rearmando el resultado con las listas resultantes a ambos lados de la cabeza.
- ▶ Su implementación:

```
qsort      :: Ord a => [a] -> [a]
qsort []   = []
qsort (x : xs) = qsort chicos ++ [x] ++ qsort grandes
  where chicos = [a | a <- xs, a <= x]
        grandes = [b | b <- xs, b > x]
```

- ▶ *Programming in Haskell*. Graham Hutton, CUP 2007.
- ▶ *Introducción a la Programación Funcional con Haskell*. Richard Bird, Prentice Hall 1997.
- ▶ “*Can Programming Be Liberated from the von Neumann Style?*”. John Backus, Turing Award Lecture. 1977
- ▶ *Why Functional Programming Matters*. John Hughes, Computing Journal. Vol 32. 1989.