Programación en Haskell

Programación Declarativa

Maximiliano Eschoyez

2021

El lenguaje Haskell

- ¿Por qué Haskell? $\rightarrow$ Presentación
- Historia sobre programación funcional
- Sitio oficial $\rightarrow$ www.haskell.org
- Bibliografía
  1. Documentación oficial
  2. Glasgow Haskell Compiler (GHC)
  3. Prelude Tour
  4. Learn You a Haskell for Great Good!
  5. RAZONANDO CON HASKELL. Un curso sobre programación funcional
- Breve intro para probar

Herramientas para trabajar

GHC Glasgow Haskell Compiler

- Haskell Platform

vscode Visual Studio Code con las extensiones

- *Haskell* (requiere GHC $\geqslant$ 8.6.4)
- *Haskell Syntax Highlighting*

Lenguaje Haskell

Se puede compilar

- Archivos estándar tiene extensión `.hs`
- Archivos *literate* tiene extensión `.lhs`

Se puede usar en forma interactiva

GHCi - Entorno Interactivo

Breve introducción para usar GHCi

- `:h` o `:help` lo obvio
- Configuraciones útiles
  - `:set +t` muestra tipos de datos
  - `:set +s` muestra estadísticas de ejecución
- `:l <arch>` o `:load <arch>` carga el archivo y lo interpreta
- `:q`, `:quit` o `Ctl+D` para salir
- No se pueden usar instrucciones multilínea directamente. Se pueden escribir
  - separadas por punto y coma

```
signo :: (Integral a) => a -> a; signo x = mod x
```

  - encerradas entre llaves

```
:{
signo :: (Integral a) => a -> a
signo x = mod x 2
:}
```

Tipos de Datos

- `Num` ⇒ Es un valor numérico
- `Real` ⇒ Es un valor numérico real
- `Fractional` ⇒ Es un valor numérico fraccional
- `Integral` ⇒ Es un valor numérico entero
  - `Int` ⇒ Limitado
  - `Integer` ⇒ Virtualmente infinito
- `Floating` ⇒ Es un valor de punto flotante
  - `Float` ⇒ Precisión simple
  - `Double` ⇒ Precisión doble
- `Bool` ⇒ Es un valor Booleano
- `Char` ⇒ Es un caracter
- `Eq` ⇒ Tiene definida la igualdad
- `Ord` ⇒ Es ordenable
- `Enum` ⇒ Es enumerable
- `Show` ⇒ Se puede mostrar como texto
- `Read` ⇒ Se puede obtener a partir de texto

Operadores Básicos

- `+` ⇒ Suma
- `-` ⇒ Resta o cambio de signo
- `*` ⇒ Multiplicación
- `/` ⇒ División
- `div` ⇒ División entera
- `mod` ⇒ División modular
- `**` ⇒ Potencia con argumentos `Floating`
- `^` ⇒ Potencia con primer argumento `Num` y segundo `Integral`
- `%` ⇒ Simplifica la relación entre dos `Integral`

Operadores Básicos

- `==` $\Rightarrow$ Igual
- `/=` $\Rightarrow$ Distinto
- `<`, `<=` $\Rightarrow$ Menor, menor igual
- `>`, `>=` $\Rightarrow$ Mayor, mayor igual
- `&&` $\Rightarrow$ Y lógico
- `||` $\Rightarrow$ O lógico

Operadores y Llamado a Funciones

- Los operadores son funciones con *definición especial*
- Los operadores son *infijos*
- Se pueden cambiar a *prefijos* usando paréntesis
- Las funciones no requieren paréntesis
- Las funciones son *prefijas*
- Se pueden cambiar a *infijas* con comillas francesas (`)

Funciones sobre Datos

- `abs` $\Rightarrow$ ...

Definición de Funciones – *por composición*

Las funciones sin restricciones se definen de forma simple por *composición*

```
Prelude> sumaDoble x y = 2 * (x + y)
Prelude> sumaDoble 6 8
28
```

Haskell infiere tipo de dato, generalmente sin inconveniente, pero conviene indicarlos con `::` y `->`

```
Prelude> :{
Prelude| sumaDoble :: Integer -> Integer -> Integer
Prelude| sumaDoble x y = 2 * (x + y)
Prelude| :}
```

Las funciones devuelven solo un resultado, siendo el último tipo de dato el tipo del resultado y los anteriores los tipos de los argumentos

Definición de Funciones – *por composición*

Haskell es *fuertemente tipado* y no permite aplicación de otros tipos de datos

```
Prelude> sumaDoble 6 8
28
Prelude> sumaDoble 6.5 7.5

<interactive>:21:11: error:
    * No instance for (Fractional Integer)
    arising from the literal '6.5'
    * In the first argument of 'sumaDoble', namely '6.5'
    In the expression: sumaDoble 6.5 7.5
    In an equation for 'it': it = sumaDoble 6.5 7.5
```

Definición de Funciones – *por composición*

Haskell permite polimofismo de tipos, se usan patrones para definir la misma función para datos de la misma clase

```
Prelude> :{
Prelude| sumaDoble :: (Num a) => a -> a -> a
Prelude| sumaDoble x y = 2 * (x + y)
Prelude| :}
Prelude> sumaDoble 6 8
28
Prelude> sumaDoble 6.5 7.5
28.0
```

Para facilitar la escritura, vamos a usar un archivo y lo leeremos en el intérprete con el comando `:load` o `:l`

```
Prelude> :l factorial.lhs
[1 of 1] Compiling Main   ( factorial.lhs, interpreted )
Ok, one module loaded.
```

Ver presentación `tema_03.pdf` y `tema_04.pdf`

Definición de Funciones – *con condicionales*

Si se necesita evaluar datos para la aplicación, la definición por composición puede hacerse *con condicionales*

```
fact'␣::␣Int␣->␣Int
fact' n = if n > 0 then n * fact'␣(n␣-␣1)
␣␣␣␣else␣if␣n␣==␣0␣then␣1␣else␣error␣"Negativo"
␣␣␣␣
```

La *indentación* indica que continua el renglón anterior

Definición de Funciones – *comparación patrones*

Una alternativa es mediante *comparación de patrones*

```
fact :: Int -> Int
fact 0 = 1
fact n = n * fact (n - 1)
```

Notar que esta versión no controla el ingreso de valores negativos

Definición de Funciones – *por partes*

Una alternativa a la *comparación de patrones* es la definición *por partes*

```
fact :: Int -> Int
factorial n
    | n == 0 = 1
    | n > 0  = n * factorial (n - 1)
    | otherwise  = error "Negativo"
```

Listas y Cadenas

- Las listas permiten agrupar datos del mismo tipo
- Las cadenas (*strings*) son listas de caracteres
- Pueden ser, virtualmente, infinitas

Operadores

- `:` $\Rightarrow$ Construcción de listas
- `++` $\Rightarrow$ Concatenación de listas
- `!!` $\Rightarrow$ Indexación de listas
- `elem` $\Rightarrow$ El elemento pertenece
- `notElem` $\Rightarrow$ El elemento no pertenece

Listas y Cadenas

Las listas se pueden definir en forma simple

```
Prelude> lista = [1, 2, 3, 4]
Prelude> lista
[1,2,3,4]
```

Pero, se construyen a partir del operador *cons* (`:`) y la lista vacía (`[]`) (asociación por derecha)

```
Prelude> lista = 1:2:3:4:[]
    => por a:[a] -> [a]
    lista = 1:2:3:[4]
    => por a:[a] -> [a]
    lista = 1:2:[3, 4]
    => por a:[a] -> [a]
    lista = 1:[2, 3, 4]
    => por a:[a] -> [a]
    lista = [1, 2, 3, 4]
Prelude> lista
[1,2,3,4]
```

Listas y Cadenas

Lo mismo ocurre con las cadenas

```
Prelude> texto = "Hola"
Prelude> texto
"Hola"
```

Pero, se construyen a partir del operador *cons* (`:`) y la lista vacía (`[]`) (asociación por derecha)

```
Prelude> texto = 'H':'o':'l':'a':[]
    => por a:[a] -> [a]
    texto = 'H':'o':'l':['a']
    => por a:[a] -> [a]
    texto = 'H':'o':['l','a']
    => por a:[a] -> [a]
    texto = 'H':['o','l','a']
    => por a:[a] -> [a]
    texto = ['H','o','l','a']
Prelude> texto
"Hola"
```

Los datos que pertenezcan a la clase `Enum` se pueden utilizar para generar enumerados

- `[m..n]` $\Rightarrow [m, m+1, \cdots, n-1, n]$
- `[m..]` $\Rightarrow [m, m+1, \cdots, \infty]$
- `[m,n..p]` $\Rightarrow [m, m+(n-m), m+2(n-m), \cdots, p-1, p]$
- `[m,n..]` $\Rightarrow [m, m+(n-m), m+2(n-m), \cdots, \infty]$

```
Prelude> [0,2..11]
[0,2,4,6,8,10]
Prelude> ['d'..'l']
"defghijkl"
```

Podemos construir nuestros enumerados a partir de `Enum`

```
Prelude> :{
Prelude| data Dia = Lunes | Martes | Miercoles
Prelude|          | Jueves| Viernes | Sabado
Prelude|          | Domingo deriving (Show, Enum)
Prelude| :}
Prelude> succ Lunes
Martes
Prelude> pred Miercoles
Martes
Prelude> [Martes .. Sabado]
[Martes,Miercoles,Jueves,Viernes,Sabado]
Prelude> [Jueves ..]
[Jueves,Viernes,Sabado,Domingo]
```

Listas Por Comprensión

Las listas se pueden definir en notación de conjuntos

```
Prelude> [x/2.0 | x <- [0..10]]
[0.0,0.5,1.0,1.5,2.0,2.5,3.0,3.5,4.0,4.5,5.0]
```

Podemos utilizar más de una variable

```
Prelude> [x+y-z | x <- [0..3], y <- [5..7], z <- [3,4]]
[2,1,3,2,4,3,3,2,4,3,5,4,4,3,5,4,6,5,5,4,6,5,7,6]
Prelude> [(x,y) | x <- [1..3], y <- [5..7]]
[(1,5),(1,6),(1,7),(2,5),(2,6),(2,7),(3,5),(3,6),(3,7)]
Prelude> [[x,y] | x <- [1..3], y <- [5..7]]
[[1,5],[1,6],[1,7],[2,5],[2,6],[2,7],[3,5],[3,6],[3,7]]
```

Y guardas para restringir los resultados

```
Prelude> [x | x <- [10..30], even x]
[10,12,14,16,18,20,22,24,26,28,30]
Prelude> [x*y | x<-[10..20], y<-[1..5], even x, x*y > 44]
[50,48,60,56,70,48,64,80,54,72,90,60,80,100]
```

Funciones sobre Listas

Funciones útiles con patrones

```
concat :: [[a]] -> [a]
concat []       = []
concat (xs:xss) = xs ++ concat xss

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

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

Funciones sobre Listas

Funciones útiles con patrones

```
zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys) = (x,y): zip xs ys
zip _      _      = []

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f _      _      = []
```

Clase IO – Imprimir Datos en Consola

Los datos de clase `Show` pueden convertirse a texto

```
Prelude> a = 2 * 5.6
Prelude> show a
"11.2"
```

La impresión en consola se realiza con `putStr` o `putStrLn`

```
Prelude> putStrLn ("Dado␣2␣*␣5.6\n" ++ "a␣=␣" ++ show a)
Dado 2 * 5.6
a = 11.2
Prelude> putStr ("Dado␣2␣*␣5.6\n" ++ "a␣=␣" ++ show a)
Dado 2 * 5.6
a = 11.2Prelude>
```

Clase IO – Leer Datos en Consola

Los datos de clase `Read` pueden obtenerse desde texto, pero debemos indicar el tipo para que se realice la conversión

```
Prelude> read "345"
*** Exception: Prelude.read: no parse
Prelude> read "345" :: Int
345
```