Introducción a Haskell
Lógica Computacional 2017-2

Lourdes del Carmen González Huesca
Roberto Monroy Argumedo
Fernando A. Galicia Mendoza

Facultad de ciencias,UNAM

Viernes, 3 de febrero del 2016

¿Qué es Haskell?

Lenguaje de programación puramente funcional multi-porpósito.
Un lenguaje de programación es un conjunto de símbolos y reglás sintácticas y semánticas que definen su estructura y el significado de sus expresiones.
Un leguaje de programación funcional, como su nombre lo indica todas sus expresiones son **funciones**.
Una **función** en `Haskell` es un mapeo que toma uno o más **argumentos** de un **tipo** y produce una **expresión** a través de una(s) **instrucción(es)**.
Un **tipo** podemos verlo como una bolsa de elementos que son creados por una definición formal.

Ventajas de Haskell

- Programas concisos.
- Listas por comprensión.
- Caza de patrones y guardias.
- Funciones de alto nivel (funciones que aceptan funciones).
- Efectos mónadicos.
- Evaluación perezosa.
- Razonamiento ecuacional.

Un ejemplo de función

Tomemos la función que dado un número entero devuelve su doble, cuya expresión en matemáticas es:

$$doble(x) = x + x$$

Cuando aplicamos la función el resultado obtenido es por medio de **sustituciones** y **simplificación** de los términos.

$$\begin{aligned} doble(4) &= 4 + 4 \ \textit{sustitución} \\ &= 8 \ \textit{simplificación} \end{aligned}$$

Intérprete

Es un programa que análiza un programa (script) y lo ejecuta.
Recordemos que los compiladores toman un programa, lo convierten a código máquina (compilación) y este ya puede ser ejecutado en la computadora.
Los interpretes toman un programa y convierten en código máquina hasta que sea necesario, es decir, hasta que el usuario solicite ejecutar una instrucción.
La principal ventaja de los programas interpretados es su facilidad de interacción con el programador, la desventaja es que son ineficientes a comparación de los programas compilados.

¿Y Haskell?

Haskell en principio es un lenguaje interpretado por su característica de ser puramente funcional, sin embargo, gracias a el tipo IO se pueden crear programas para que sean compilados.
Los dos interpretes más populares de Haskell son: GHC y Hugs.
Hugs es un interprete más pequeño, ya que su uso es principalmente para la educación.
Claramente vamos a utilizar GHC.

GHC

The Glasgow Haskell Compiler por sus siglas en inglés (o como sus propios autores le denominaron *The Greatest Haskell Compiler*) es un compilador e interprete del lenguaje de programación `Haskell`.
Para abrir el interprete, solo deben abrir la terminal y escribir la orden:
`ghci`
Una vez abierto mostrará la siguiente expresión:
`Prelude>`
Este se le llama *prompt* y es donde le indicaremos al interprete que acciones debe realizar.

Jugando con el interprete

Los comandos básicos para el interprete son:

- `:l <ruta_Archivo>`: Carga un programa al interprete.
- `:r`: Teniendo un *script* abierto, lo vuelve a cargar.
- `:h [Comando]`: Muestra algunos comandos y una breve descripción de estos.
- `:cd <ruta_Nueva>`: Cambia el directorio actual de trabajo.
- `:t <Expresion>`: Devuelve el tipo de la expresión dada.
- `:q`: Cierra el interprete.

El preludio

`Prelude` es la biblioteca donde se encuentran las funciones predefinidas y tipos básicos del lenguaje sin necesidad de importar otras.
Algunos tipos que podemos trabajar en el preludio son: números (enteros y reales), listas, cadenas de texto, tuplas, etc.
Y tenemos funciones básicas para estos tipos.

Ejemplo

```
Prelude> :t 3
3 :: Num a => a
Prelude> :t 3.0
3.0 :: Fractional a => a
Prelude> :t 'a'
'a' :: Char
Prelude> :t "a"
"a":: [Char]
Prelude> :t (1,2)
(1,2) :: (Num t, Num t1) => (t, t1)
```

Mas ejemplos de Prelude

Ejemplo

```
Prelude> :t (+)
(+) :: Num a => a -> a -> a
Prelude> :t (/)
(/) :: Fractional a => a -> a -> a
Prelude> :t id
id :: a -> a
Prelude> 3+2
5
Prelude> 3.0+2
5.0
Prelude> (3+2) == (3.0+2)
True
Prelude> abs (-1)
1
```

Buenas prácticas

Cualquier lenguaje de programación cuenta con un estándar llamado **buenas prácticas**. Este estándar es un conjunto de reglas y convenios a la hora de programar, su finalidad es tener código limpio para una fácil lectura y en caso de ser necesario una rápida actualización.
En el caso de `Haskell` tenemos las siguientes buenas prácticas:

- Todo script debe empezar con una mayúscula, no tener espacios y ser de nombre corto.
- Toda función debe tener firma y empezar con mínuscula.
- Toda función debe seguir el siguiente comentado:

```
--| Nombre de la funcion. Descripcion de la funcion.
--
-- → Ejemplos de aplicacion de la funcion.
```

- Todo tipo definido debe iniciar con mayúscula y seguir el siguiente comentado:

```
--| Nombre del tipo. Descripcion de lo que representa.
```

- Un estándar que adoptaremos es al inicio de cada script, ingresar el siguiente comentado:

```
--| Logica Computacional 2017-2
--| Numero y titulo de practica
--| Profesor: Dra. Lourdes del Carmen Gonzalez Huesca
--| Ayudante: Roberto Monroy Argumedo
--| Laboratorio: Fernando A. Galicia Mendoza
--| Integrantes:
--| Integrante1 No.Cuenta Email
--| Integrante2 No.Cuenta Email
```

- Es opcional utilizar acentos, depende del gusto de cada uno.

Estructura de un archivo

Un archivo `Haskell` debe tener la siguiente estructura:

```
--Comentarios iniciales

module Nombre_Archivo where

--Tipos definidos

--Funciones principales

--Funciones auxiliares

--Ejemplos
```

Tipos de datos

Un *tipo* es una forma de identificar un dato, por ejemplo un entero en `Haskell` tiene asignado el tipo `Int`.
`Haskell` proveé un tipo de datos llamados *primitivos*, su nombre es debido que son los elementos más básicos para poder programar, con este tipo de datos uno puede definir nuevos tipos.
Algunos tipos primitivos son:

- `Int`: Representa los números enteros.
- `Float`: Representa los números reales (punto flotante).
- `Double`: Representa los números enteros (doble precisión).
- `Char`: Representa los carácteres.
- `String`: Representa las cadenas de texto.
- `(a1,a2,...,an)`: Representa colecciones de datos de un tamaño específico (tuplas).
- `[a1,a2,...]`: Representa colecciones de datos sin un tamaño específico (listas).

Operaciones aritmético-booleanas

- `+` Suma.
- `-` Resta.
- `*` Producto.
- `^` Potencia en enteros.
- `**` Potencia en reales.
- `div` División enteros.
- `/` División en reales.
- `<` Menor que.
- `<=` Menor o igual que.
- `>` Mayor que.
- `>=` Mayor o igual que.
- `==` Igualdad.

Comodidad matemática

En matemáticas una función tiene la siguiente estructura:

$$f : A \to B$$
$$f(x) = y$$

Y en `Haskell` la estructura de una función es:

```
f :: A → B
f x = y
```

Tuplas

En cualquier lenguaje de programación una estructura es una manera abstracta de definir objetos, estructuras matemáticas, etc.
Una estructura escencial en `Haskell` es la tupla, esta estructura nos da una forma de poder representar ordenadas de una forma sencilla.
Por ejemplo si queremos representar la siguiente función en `Haskell`.

$$g : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}^2$$
$$g((x_1, y_1), (x_2, y_2)) = (x_1 + x_2, y_1 + y_2)$$

En `Haskell` se escribe:

```
g : (Double,Double) → (Double,Double) → (Double,Double)
g (x1,y1) (x2,y2) = (x1+x2,y1+y2)
```

Expresión `let` y cláusula `where`

Supongamos que queremos utilizar una expresión mas de una vez en una función, esto resultaría tedioso al momento de programar.
La expresión `let` nos permite definir variables para poder utilizarlas dentro de una función:

```
let x = <expresion> in <expresion con variable x>
```

Otra forma de hacer lo anterior es utilizando la cláusula `where`.

```
<expresion con variable x> where
        x = <expresion>
```

La cláusula case

`Haskell` proveé de una expresión de caza de patrones llamada cláusula `case`, donde tiene dos expresiones como parámetro, el término a analizar y los resultados por cada subtérmino obtenido.
Por ejemplo la función suma se puede definir de la siguiente forma:

```
suma :: Nat → Nat → Nat
suma x y = case (x,y) of
(C,y) → y
(S x,y) → S (suma x y)
```

Si observamos esta última analizamos los casos sobre el par que constituyen los términos a sumar.

Sinónimos

Escribir `(Double,Double)` puede resultar cansado a la hora de programar, entonces podemos darle un sinónimo para que sea más fácil su escritura. Tomando el ejemplo de $\mathbb{R}^2$, podemos agregar la expresión:

```
type R2 = (Double,Double)
```

Por lo que nuestra función g, la podríamos definir como:

```
g : R2 -> R2 -> R2
g (x1,y1) (x2,y2) = (x1+x2,y1+y2)
```

Gramáticas en Haskell

Una gramática es una estructura matemática con un conjunto de reglas para la formación de palabras.
Una gramática la denotamos de la siguiente forma:

$$S ::= < atomo > \mid Constructor\ S \ldots S$$

Haskell nos proveé de una instrucción para definir gramáticas de forma sencilla:

```
data S = <atomo> | Constructor S ...S
```

Los números naturales

Consideremos por ejemplo la siguiente gramática:

$$Nat \;=\; C \mid S\ Nat$$

Entonces en `Haskell` queda implementado de la siguiente forma:

```
data Nat = C | S Nat deriving(Show)
```

La expresion `deriving(Show)` se utiliza para imprimir cadenas de texto, tal y como se ve en pantalla.
Mas adelante veremos como indicarle a `Haskell` como queremos que imprima nuestros elementos resultantes de los tipos de datos que definamos.

Funciones recursivas

La definición recursiva de tipos de datos permite definir funciones sobre los mismos utilizando una técnica de caza de patrones.
Por ejemplo la suma de los números naturales se define de la siguiente forma en `Haskell`:

```
suma :: Nat → Nat → Nat
suma C y = y
suma (S x) y = S (suma x y)
```

Aquí la caza de patrones se hace directamente sobre los parámetros de la función, veamos otra forma de hacer caza de patrones mas elegante.

Listas

Una lista es una secuencia de elementos de un mismo tipo, cuyos elementos están dentro de dos corchetes y separados por comas.

Ejemplo (Ejemplos de listas)

- `[1,2,3,4]` - *Lista de números enteros.*
- `[1.23,134.6,0.1111119]` - *Lista de números relaes.*
- `[(1,2),(5,6),(9,12)]` - *Lista de tuplas de números enteros.*
- `[[1,2,3],[3,4,5]]` - *Lista de listas de números enteros.*
- `[([1,2,3],[4,5,6])]` - *Lista de tuplas de listas de números enteros.*

Listas vs Tuplas

Observemos que las listas son un medio de almacenamiento para datos. . . y las tuplas también.
La diferencias principales son las siguientes:

- Las listas pueden tener una cantidad **infinita** de elementos, las tuplas no.
- Las tuplas pueden tener **varios tipos** de elementos, las listas no.

Ejemplo

- *`[0..]` - Lista de todos los números enteros.*
- *`(1,1.01,'a',"Hola")` - Tupla conformada por un entero, un real, un carácter y una cadena de texto.*

Listas y tuplas. . . somos invencibles

A pesar de parecer dos estructuras muy simples, con una buena formalización de un problema estos tipos nos pueden ser de gran utilidad al definir nuevos tipos.
Por ejemplo:

- `(Int,Int)`: Representamos los números racionales.
- `(Double,Double)`: Representamos los números complejos.
- ¿Qué otros ejemplos se nos ocurren?