ANÁLISIS, VALIDACIÓN Y DEPURACIÓN DE SOFTWARE

Práctica 1
Recordando Haskell

Alicia Villanueva

Curso 2017/2018

Índice

1. Objetivo de la práctica

El objetivo de la práctica es el de recordar el entorno de programación funcional ghci del lenguaje Haskell. El objetivo no es el de aprender a programar en Haskell, sino revisar los conceptos básicos (ya aprendidos en otras asignaturas) y que nos servirán más adelante. Puede encontrarse material de apoyo adicional en `http://www.haskell.org`.

2. Trabajo a realizar y evaluación

1. Lee el boletín y haz los ejercicios propuestos.
2. Guarda tus ejercicios en el disco local para tenerlos disponibles en la próxima sesión.

Al finalizar la sesión deberás completar un test individual sobre los contenidos y ejercicios en este boletín.

3. Programación funcional

En programación funcional, todos los programas son funciones. Podemos escribir muchas funciones en un mismo archivo. Las funciones pueden invocarse entre sí.

Toda expresión (y toda función) tiene un tipo asociado, que podemos escribir explícitamente antes de su definición. El tipo de una función consiste en especificar los tipos de sus argumentos de entrada y el de la salida. Después del tipo, una serie de ecuaciones definen el comportamiento de la función. Las funciones siempre tendrán una única salida. Cuando queramos que una función devuelva dos valores, tendremos que recurrir al uso de tuplas, de forma que cada elemento de la tupla representará uno de los valores resultado. Las tuplas se definen usando paréntesis y separando en su interior los elementos por comas.

En cuanto a la sintaxis, para definir el **tipo de una función** escribiremos su nombre, a continuación `::` y después los tipos de los argumentos de entrada separados por `->`. El último tipo que aparece en la lista de argumentos siempre corresponde al tipo del resultado de la función.

Ejemplo 1 `increment :: Int -> Int` *define el tipo de una función que toma como argumento de entrada un entero y da como resultado otro entero.*

Para definir el comportamiento de la función escribimos:

```
increment x = x + 1
```

donde la variable `x` *es el parámetro formal (argumento de entrada) de la función, y la expresión a la derecha del símbolo = representa el resultado de la función.*

Ejercicio 1 *Crea un fichero de texto llamado* `Increment.hs` *e introduce el siguiente programa:*

```
module Increment where
increment :: Int ->Int
increment x = x + 1
```

Guarda los cambios y lanza en la línea de comandos el intérprete con el comando ghci. *Aparecerá en pantalla algo parecido a:*

```
GHCi, version 7.8.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude>
```

Carga el fichero introduciendo a continación la orden `:l Interprete.hs`. *El comando* `:l` *es equivalente a* `:load` *que carga un fichero en el intérprete. A partir de este momento, una vez cargado tu función, puedes invocarla desde la línea de comandos. Escribe por ejemplo* `increment 4`.

`:load` (o bien `:l`) carga un fichero Haskell en el intérprete. **`:add`** carga un módulo adicional en el intérprete. **`:type`** nos dice de qué tipo es una expresión dada. **`:quite`** (o bien `:q`) sale del intérprete. **`:?`** muestra los comandos disponibles.

El programa del ejemplo anterior podría *leerse* como "*increment* es una función que toma como entrada un número entero y devuelve como resultado otro número entero. Además, siendo x el entero de entrada (*increment x*), la función devuelve como salida el resultado de hacer la operación $x + 1$".

Un ejemplo de tipo de función que devolvería un par de valores sería:

`roots :: (Float, Float, Float) -> (Float, Float)`,
la cual toma como entrada *un único* parámetro (una tupla de tres elementos) y devuelve otro parámetro: una tupla de dos elementos.

El tipo `Bool` define los valores `True` y `False`. Para calcular el resultado de una función podemos usar la expresión

`if` *cond* `then` *result1* `else` *result2*

donde *cond* es una condición booleana y las ramas definen el resultado en los casos en los que la condición sea cierta y falsa respectivamente. En las condiciones podemos usar operadores booleanos `&&` (conjunción), || (disyunción) y `not` (negación).

Ejercicio 2 *define una función que, dados dos valores enteros, dé como resultado si los dos valores son iguales o no. Utiliza la expresión condicional* if-then-else. *El operador de comparación de igualdad en* Haskell *es* $==$.

Las funciones se pueden definir mediante casos por *pattern matching*. Con esta estrategia se identifican los casos usando patrones en los argumentos de forma que definiremos el resultado en la parte derecha del igual cuando el patrón de la parte izquierda se satisface. Podemos usar variables *anónimas* escribiendo un guión bajo _ cuando el argumento de entrada no se use en la parte derecha de la función.

$$\begin{aligned} &f\ formal\_parms = exp1 \\ &f\ formal\_parms = exp2 \\ &... \\ &f\ \_ \qquad\qquad\quad = exp3 \end{aligned}$$

Ejercicio 3 *Define la función que, dado un entero, devuelve cierto en caso de que sea igual a 0 y falso en caso de que sea cualquier otro valor. Se debe usar la estrategia de* pattern matching.

Una tercera estrategia es la de usar condiciones sobre los argumentos que distingan los casos. En este caso, tendremos una sola regla y dentro de ella especificaremos los casos separados por |. Tras cada | definiremos una condición booleana en base a los argumentos de entrada y su respectivo resultado tras el =. Podemos usar como condición la palabra clave otherwise en la última opción. El esquema general sería:

$$\begin{aligned} f\ formal\_parms \mid & cond1 = exp1 \\ \mid & cond2 = exp2 \\ & \vdots \\ \mid & \text{otherwise} = expn \end{aligned}$$

Ejercicio 4 *Reescribe la misma función que en el ejercicio anterior pero usando el esquema de las guardas.*

La sentencia `where` permite declarar valores de variables de forma local. Es útil cuando el valor de una variable se va a usar varias veces. Hay que tener cuidado con el sangrado ya que todas las declaraciones locales que se hagan deben estar al mismo nivel de sangrado.

```
roots (a,b,c) = (x1, x2) where
                x1 = e + sqrt d / (2 * a)
                x2 = e - sqrt d / (2 * a)
                d = b * b - 4 * a * c
                e = -b / (2 * a)
```

Diremos algo más sobre la **sangría**. Las declaraciones de más alto nivel (las declaraciones de funciones) se harán en la primera columna del texto y terminan en la siguiente declaración que empiece en la primera columna de texto. Además, si queremos continuar una expresión en la siguiente línea lo podremos hacer siempre y cuando la sangría de la segunda línea sea superior a la anterior.

Podemos utilizar **comentarios** introducidos por dos guiones `--` para comentar una línea o usar `{-` y `-}` para comentar el bloque de código entre ellos. El caracter \ actúa como **caracter de escape**, haciendo que el caracter inmendiatamente sucesivo sea interpretado de forma literal y no con el significado que tendría según la semántica de Haskell.

Además, cuando queramos invocar funciones de **forma infija** en vez de prefija, deberemos poner delante y detrás del nombre de la función, una comilla backquote (es decir, el acento grave). Esta operación puede verse como una transformación de una función en un operador. La transformación contraria también es posible. Si tenemos un operador (por ejemplo la suma +), podemos usarla como función si la encerramos entre paréntesis. De estaforma, `(+)` será la notación prefija para la suma.

Fíjate en los siguientes ejemplos:

- La expresión `div 3 4` donde `div` es una función con dos parámetros, se puede escribir también en forma infija como `3 `div` 4`.
- La expresión `3 + 4` donde + es el operador suma se puede escribir también en forma prefija como `(+) 3 4`.

Otros **tipos predefinidos** en Haskell:

`Bool`	True, False
`Int`	-100, 1, 2, ...
`Integer`	-3333333333, 3, 4843984909873, ...
`Float/Double`	-3.22425, 0.0, 3.0, ...
`Char`	'a', 'z', ';', ...
`String`	"Hola", "Funcion", ...

Mencionaremos también cuáles son los **operadores de comparación** de Haskell: == para la igualdad, /= para la desigualdad, < para la relación menor que, <= para la relación menor o igual, > para el mayor y >= para el mayor o igual. Estos operadores están definidos para caracteres y para tipos numéricos.

3.1. Listas

Las listan nos proporcionan un mecanismo para especificar estructuras con un número indefinido de elementos *del mismo tipo*. Se definen especificando el tipo de la lista entre corchetes (ej: lista de enteros: `[Int]`). Puede no tener ningún elemento, en cuyo caso la denotamos como `[]`. Los elementos dentro de una lista se separan con comas (`[1,2,3]`). Los *strings* también son listas, en concreto son listas de *Chars*. Estructuralmente podemos acceder al primer elemento de una lista (la cabeza) o al resto de la lista (la cola) mediante la expresión `x:xs`. Es decir, `x` representa la cabeza y `xs` la cola, pudiendo ser esta última la lista vacía. Para poder acceder al resto de elementos de una lista de forma individual, deberemos hacerlo de forma recursiva sobre la lista (aunque también hay operadores predefinidos para hacerlo).

Las funciones `head` y `tail` se comportan como sigue: dada una lista, proporciona el primer elemento o la cola de la lista respectivamente. Otras funciones definidas sobre listas que mencionaremos pero no explicaremos son `take`, `drop`, `takeWhile` y `dropWhile`.

La recursión es el recurso principal para la programación funcional. Deberemos identificar el caso base y el caso recursivo de las funciones. Suele resultar útil pensar en una ejecución concreta, en qué se hace en un paso determinado y qué resultados necesita de la iteración anterior.

Ejercicio 5 *Escribe una función que, dada una lista y un número entero, devuelva el elemento de la lista que se encuentra en la posición indicada por el entero. Identifica el caso base y el paso de recursión antes de comenzar a escribir.*

Dos esquemas que usaremos frecuentemente:

- cuando trabajemos con recursiones sobre números, normalmente el caso base será el caso en el que el valor es 0 (y/o algunas veces 1). El caso recursivo se tratará del sucesor de un número natural dado,
- cuando trabajemos con listas, normalmente el caso base será el caso en el que la lista es la lista vacía (y/o algunas veces la lista con un elemento) y el caso recursivo trabajará con la cola de la lista.

Funciones básicas para listas:

`head`	devuelve el primer elemento de una lista
`tail`	devuelve la cola de la lista o borra el primer elemento de una lista
`length`	devuelve la longitud de una lista
`reverse`	devuelve la lista invertida
`++`	concatena dos listas
`map`	aplica una función a cada elemento de la lista
`filter`	devuelve todos los elementos de la lista que satisfacen una condición
`foldr`	combina los elementos de una lista con una función especificada y un valor inicial
`zip`	combina dos listas obteniendo una lista de pares a partir de ellas
`zipWith`	combina los elementos de dos listas aplicándoles una función determinada
`concat`	dada una lista de listas, obtiene la concatenación de todas ellas

Las funciones pueden tomar funciones como argumentos. Una de las más importantes es la función predefinida map. Trata de recordar cómo funcionaba a partir de su definición de tipos y de su implementación:

```
map :: (a -> b) -> [a] -> [b]

map f [] = []
map f (x:xs) = f x : map f xs
```

Ejercicio 6 *Sabiendo que existe una función* `ord` *(disponible al importar el módulo* `Char`[1]*) que toma como argumento un* `Char` *y devuelve el código ASCII que lo representa, define una función que tome como valor de entrada un* `String` *y devuelva la lista de códigos ASCII que lo representan.* ***Se debe usar la función map.***

`foldr` procesa los elemenos de una lista de forma acumulada usando una función binaria. Por ejemplo, `foldr (*) 1 [1,4,3,5,2]` devolverá como resultado el producto de los elemenso de la lista, siendo el caso base 1, es decir, cuando no le queden más elementos en la lista aplicará la operación con el 1 ya que es el segundo argumento de la función. En particular hará la siguiente operación: `1 * (4 * (3 * (5 * (2 * 1))))`.

3.2. Composición de funciones

Podemos componer funciones de forma muy natural. En matemáticas tenemos dos posibles notaciones para la composición de funciones: $f(g(x))$ o $(g \circ f)(x)$. La idea clave es que el resultado de la función g debe ser del mismo tipo que el argumento de la función f para que la composición funcione.

En Haskell la composición de funciones se especifica de la siguiente forma: `(f . g) x`, o bien `f (g x)`.

3.3. Tipos de datos algebraicos

Para representar tipos de datos de una forma más intuitiva y estructurada podemos recurrir al uso de *sinónimos*. Por ejemplo, en el ejemplo de la función `roots` mostrado anteriormente podríamos haber definido los tres elementos de entrada y los dos de salida como sigue:

```
type Poly2 = (Float, Float, Float)
type Roots2 = (Float, Float)

roots :: Poly2 -$>$ Roots2
```

Pero en vez de limitarnos a usar abreviaturas para tipos, podemos también definir *nuevos tipos* algebraicos de datos usando la expresión data. La estructura general es:

[1] Si usas un ghci basado en Haskell2010, debes importar `Data.Char`

data *nombreTipo* = *constructorDatos1 tiposArgumentos1*
| *constructorDatos2 tiposArgumentos2*
...

Un ejemplo de tipo algebraico de datos sería la definición de los naturales `Nat` expresados con notación de Peano:

```
data Nat = Cero | Suc Nat
```

Donde se definen los valores del tipo de datos `Nat` como el conjunto de valores `Cero`, `Suc Cero`, `Suc (Suc Cero)`,.... Es decir, valores que concuerdan con la definición recursiva dada.

Para evitar problemas a la hora de mostrar valores por pantalla de los nuevos tipos, es recomentable añadir al final de la definición del nuevo tipo las palabras clave `deriving Show`.

Por ejemplo, podemos definir un nuevo tipo de datos `Libro`:

```
data Libro = Text String String Int deriving Show
```

Donde un valor del nuevo tipo podría ser

```
Text "Calderón de la Barca" "La vida es sueño" 1635
```

3.4. Polimorfismo

Es posible definir funciones polimórficas. Se caracterizan por el uso de variables de tipo en la definición del perfil. Un ejemplo lo hemos visto ya cuando hemos visto la definición de la función map.

Por ejemplo, el perfil

```
suma :: a ->a ->a
```

denota que la función suma toma dos parámetros del mismo tipo (veremos más adelante cómo restringir ese tipo a un tipo numérico) y devuelve como resultado un valor del mismo tipo que los valores de entrada.

3.5. Clases de tipos

En Haskell los tipos de datos se organizan en una jerarquía de clases que los agrupan en función de las operaciones que pueden realizarse sobre los valores del tipo. Los tipos de datos pueden ser, por lo tanto, instancias de una clase determinada, lo que supone que hay ciertas operaciones implementadas para ese tipo. Las clases pueden tener una implementación por defecto de las operaciones, o bien se puede redefinir la implementación de las operaciones cuando se define el tipo como instancia de la clase.

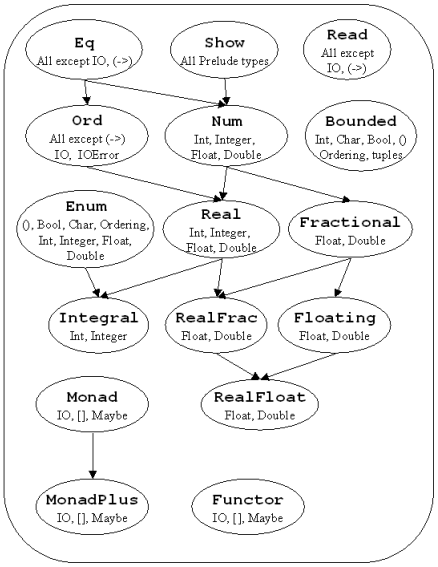

Figura 1: Jerarquía de clases en Haskell

Existen algunas clases predefinidas y los tipos predefinidos son instancia de algunas de esas clases. En el ejemplo de `suma` anterior, para asegurarnos de que no se define para cualquier tipo de datos, sino para un tipo numérico, escribimos:

```
suma :: Num a => a -> a -> a
```

Lo que aparece a la izquierda del símbolo `=>` son las condiciones impuestas sobre el tipo `a`. En este caso, se exige que sea una instancia de la clase `Num`.

Clases que aparecen frecuentemente:

Clase `Eq` Contiene los tipos para los que la igualdad `==` y desigualdad `/=` están definidas (como ejemplo, son instancias de esta clase los tipos predefinidos `Int` y `Char`).

Clase `Ord` Funciones que se definen: `<`, `>`, `>=`, `<=`, `max` y `min`. Todos los tipos predefinidos excepto `IO`, `IOError` y `->` son instancia de esta clase.

Clase `Num` Funciones que se definen: `+`, `-`, `*`, `negate`, `abs`, `signum`, ... Ejemplos de tipos instancia de esta clase: `Int`, `Integer`, `Float`, `Double`, `Ratio`.

Clase `Show` Se define la función `show` y todos los tipos predefinidos excepto `IO` y `->` son instancia de esta clase.

El diagrama de la Figura 1 muestra la jerarquía de clases de Haskell.

Como ejemplo, podemos definir una clase nueva de la siguiente forma:

```
class Stats a where
```

```
   media :: [a] -> a
   minList :: [a] -> a
```

Donde se está definiendo una nueva clase donde los tipos que sean instancia de la misma deberán tener definidas las operaciones de `media` y `minList`. Podemos crear la clase de forma que dé una implementación por defecto de dichas funciones:

```
class (Num a) => Stats a where
   media :: [a] -> a
   minList :: [a] -> a
   media xs = foldr (+) 0 xs
```

Donde se da implementación por defecto para una de las funciones. Para poder dar esta implementación se ha debido restringir el tipo paramétrico a aquéllos tipos que sean instancia de la clase `Num`, ya que en la definición se usa la función `(+)`.

Volvamos ahora al tipo algebraico `Nat` definido anteriormente. Es un tipo de datos nuevo que no es instancia de ninguna clase. Si quisiéramos hacerlo instancia de la clase `Num`, podríamos hacerlo de la siguiente forma:

```
instance Num Nat where
  Cero + x = x
  (Suc x) + y = Suc (x+y)
  Cero * x = Cero
  (Suc x) * y = y + (x*y)
```

De esta forma tendríamos las funciones `(+)` y `(*)` sobrecargadas para trabajar, no sólo con los tipos predefinidos sino también con el nuevo tipo `Nat`.

Un tipo puede ser instancia de más de una clase. Supongamos que queremos representar los naturales escritos en notación de peano con su correspondiente valor de `Int`. Para ello, podríamos definir el tipo como instancia de la clase `Enum` (las funciones características de esta clase son `toEnum` y `fromEnum` que pasan de entero a tipo enumerado y viceversa):

```
instance Enum Nat where
  fromEnum Cero = 0
  fromEnum (Suc x) = 1 + (fromEnum x)
```

Referencias

[1] *Haskell-Tutorial*. Damir Medak and Gerhard Navratil, 2003.

[2] *Yet Another Haskell tutorial*. HAl Daume III, 2002. `http://www.cs.utah.edu/~hal/docs/daume02yaht.pdf`

[3] *Introducción a la Programación Funcional con Haskell*. R. Bird, Prentice-Hall, 2000.