Universidad de Huelva

Práctica 2

Tipos y funciones básicas

Índice

2.1 Características básicas del sistema de tipos

- Haskell es un lenguaje **fuertemente tipado**
  - Esto quiere decir que toda expresión tiene un tipo de dato concreto y que no puede modificarse de manera implícita. Por ejemplo, si una función está definida sobre un tipo de dato entero e intentamos llamarla con una expresión de tipo *string*, el compilador generará un error de tipo.
  - Haskell no realiza conversiones de tipo implícitas (*casting* o *coercion*). Si queremos transformar el tipo de dato de una expresión es necesario aplicar un operador de conversión de tipos.
  - El lenguaje C, por ejemplo, realiza transformaciones automáticas entre *int* y *float* o entre *int* y *double*. Se puede llamar a una función definida sobre *float* con un valor *int* porque el compilador de C realiza una conversión automática. En Haskell esto no ocurre.

- Haskell es un lenguaje con **tipado estático**
  - Esto quiere decir que el tipo de dato de cualquier expresión es conocido en tiempo de compilación.
  - La propiedad contraria se conoce como tipado dinámico y es propia de otros lenguajes como Python o Rubi. En estos lenguajes el sistema de tipos es mucho más flexible y permisivo y solo se genera un error en tiempo de ejecución cuando se intenta operar sobre un objeto que no tenga implementada la función requerida.

- Haskell utiliza **inferencia de tipos**:
  - Esto quiere decir que en la mayoría de los casos el programador no está obligado a indicar explícitamente el tipo de dato de las expresiones sino que el compilador es capaz de inferirla considerando los tipos de datos básicos incluidos y las funciones utilizadas en las expresiones.
  - En lenguajes como C o Java es necesario declarar el tipo de dato de cada variable a utilizar o el tipo de dato de los argumentos de las funciones. En Haskell es el compilador el que infiere el tipo de dato estudiando las expresiones vinculadas a esas variables.
  - Haskell permite indicar los tipos de datos (*type signature*), pero por defecto el compilador intentará asignar a las expresiones el tipo de dato más general posible que sea compatible con la expresión.
  - Por ejemplo, la función ***head*** devuelve el primer elemento de una lista, sea cual sea el tipo de dato de los elementos de la lista.

- Haskell utiliza **tipos de datos jerárquicos (type classes)**:
  - Las *type classes* permiten definir una jerarquía de tipos en la que las subclases contienen las mismas funciones que la clase superior y puede añadir nuevas clases.
  - Las *type classes* desarrollan herencia múltiple (en cierto sentido pueden compararse a las interfaces de Java).
  - Por ejemplo, la clase *Eq* describe cualquier tipo de dato que tenga definido el comportamiento sobre los operadores "==" y "/=".
  - Se pueden definir nuevas clases que extiendan a otras utilizando la clausula **deriving.**
  - Al definir tipo de dato se puede indicar que es una instancia de una o varias type classes por medio de la clausula **instance**.

- El operador de definición de tipo (coercion) es "::".
- El intérprete de Haskell permite utilizar el comando **:type** para obtener el tipo de dato inferido en una expresión. Por ejemplo:

```
Prelude> :type 2 + 8
2 + 8 :: Num a => a

Prelude> :type 2
2 :: Num p => p

Prelude> :type pi
pi :: Floating a => a

Prelude> :type 1.5
1.5 :: Fractional p => p
```

```
Prelude> :type 'A'
'A' :: Char

Prelude> :type "Hola mundo"
"Hola mundo" :: [Char]

Prelude> :type True
True :: Bool

Prelude> :type sqrt 2.0
sqrt 2.0 :: Floating a => a
```

- Se puede indicar al intérprete que muestre siempre la información del tipo con el comando **:set +t**

```
Prelude> :set +t
Prelude> 2+8
10
it :: Num a => a
```

- Para obtener información sobre un tipo de dato se utiliza el comando **:info**

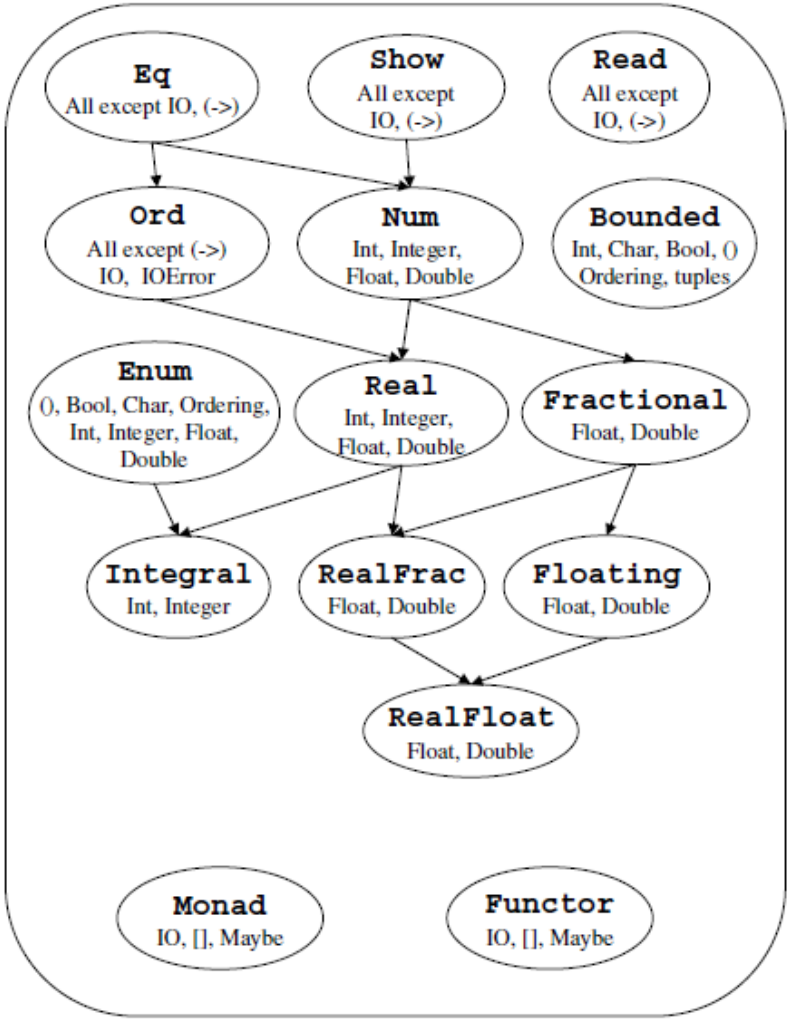
```
Prelude> :info Eq
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  {-# MINIMAL (==) | (/=) #-}
```

Los siguientes tipos están definidos en el módulo Prelude

- **Bool**:
  - Tipo de dato booleano. Admite dos valores: **True** y **False**.
- **Ordering**:
  - Tipo de dato que responde al valor de una comparación. Admite los valores **LT**, **EQ** y **GT**.
- **Char**:
  - Describe un carácter en formato unicode.
- **Int**:
  - Describe un tipo de dato entero con el tamaño estandar del procesador (32 o 64 bits). El estandar solo garantiza que debe tener más de 28 bits.

- **Integer**:
  - Describe un valor entero no acotado. Este tipo de dato permite realizar operaciones con una precisión ilimitada aunque las operaciones son computacionalmente más costosas.
- **Float**:
  - Tipo de dato en coma flotante en formato IEEE de 32 bits. El estandar de Haskell recomienda utilizar mejor el formato Double.
- **Double**:
  - Tipo de dato en coma flotante en formato IEEE de 64 bits.
- **()**:
  - Denominado *unit* refleja una tupla de 0 elementos. Se puede considear un tipo de dato similar a *void* en otros lenguajes.

- Clases estandar contenidas en Prelude:



- **Eq**:
  - Clase que describe tipos de datos sobre los que están definidos los métodos de igualdad (==) y desigualdad (/=).
- **Ord**:
  - Clase que describe tipos de datos que pueden ordenarse. La clase define los métodos *compare* (que genera un resultado de tipo *Ordering*), *min* (que obtiene el mínimo entre dos valores), *max* (que obtiene el máximo entre dos valores) y los operadores (>), (>=), (<) y (<=) que devuelven valores de tipo *Bool*.
- **Show**:
  - Clase que describe tipos de datos cuyos valores se pueden mostrar como cadenas.

- **Read**:
  - Clase que describe tipos de datos cuyos valores se pueden parsear desde una cadena de caracteres.
- **Enum**:
  - Clase que describe tipos de datos que se pueden enumerar. Contiene los métodos *succ* y *pred* que obtiene el sucesor y predecesor de un valor. También contiene los métodos *toEnum* y *fromEnum* (que obtiene el valor a partir del índice entero) y *enumFrom*, *enumFromThen*, *enumFromTo* y *enumFromThenTo* (que construyen listas a partir de valores iniciales y finales).
- **Bounded**:
  - Clase que describe tipos de datos acotados. Contiene los métodos *maxBound* y *minBound* que devuelven los valores máximos y mínimos del tipo.

- **Num**:
  - Clase que describe todos los tipos de datos numéricos. Contiene los métodos (+), (*), (-) y *negate*, que puede escribirse como el prefijo (-).
- **Real**:
  - Clase que describe números reales (tanto enteros como en coma flotante). Contiene el método *toRational* que transforma los datos numéricos a formato racional (en forma de fracción de enteros).
- **Integral**:
  - Subclase de *Num* y *Real* que describe datos enteros (sin decimales). Contiene los métodos *quot* (división entera truncada hacia el 0), *rem* (resto de la división entera truncada hacia el 0), *div* (división entera truncada hacia menos infinito) y *mod* (resto de la división entera truncada hacia menos infinito).

- **Fractional**:
  - Subclase de *Num* que describe datos en coma flotante. Añade el método (/) que describe la división real así como los métodos *recip*, que calcula el inverso o recíproco, y *fromRational* que transforma una fracción a formato en coma flotante.
- **RealFrac**:
  - Subclase de *Real* y *Fractional* que añade métodos de redondeo. El método *properFraction* obtiene la parte entera y la parte decimal y los métodos *truncate*, *round*, *ceiling* y *floor* desarrollan diferentes formas de redondeo.
- **Floating**:
  - Subclase de *Fractional* que añade las funciones matemáticas básicas sobre números reales: *pi*, *exp*, *log*, *sqrt*, **, *logBase*, *sin*, *cos*, *tan*, *asin*, *acos*, *atan*, *sinh*, *cosh*, *tanh*, *asinh*, *acosh*, *atanh*.

- **RealFloat**:
  - Subclase que añade métodos de operación a nivel de bit con los datos en coma flotante: *floatRadix*, *floatDigits*, *floatRange*, *decodeFloat*, *encodeFloat*, *exponent*, *significand*, *scaleFloat*, *isNaN*, *isInfinite*, *isDenormalized*, *isNegativeZero*, *isIEEE*.
- **Monad**:
  - Clase que encapsula una forma de combinar cálculos. Una mónada necesita un constructor, un función de combinación (*bind* o >>=) y una función *return*. La mónada *IO* se utiliza para definir operaciones de entrada y salida. Las mónadas se usan también como forma de encapsular funciones impuras en Haskell.
- **Functor**:
  - Clase que describe un tipo que puede ser mapeado.

2.3 Funciones aritméticas y lógicas básicas

- Operaciones lógicas
  - (&&) :: Bool -> Bool -> Bool -- AND con cortocircuito
  - (||) :: Bool -> Bool -> Bool -- OR con cortocircuito
  - not :: Bool -> Bool -- NOT
- Operaciones aritméticas
  - (+) -- Suma definida sobre tipos Num
  - (-) -- Resta definida sobre tipos Num
  - (*) -- Producto definido sobre tipos Num
  - (/) -- División exacta sobre tipos Num generando Fractional
  - (^) -- Potencia de un Num a un exponente entero
  - (^^) -- Potencia de un Fractional a un exponente entero
  - (**) -- Potencia de un Fractional a un exponente Fractional

- Las funciones binarias pueden escribirse en formato infijo si se escribe el nombre entre comillas `.

```
Prelude> quot 25  3
8
Prelude>  25 `quot` 3
8
```

- Funciones de división entera:
  - quot -- División entera con redondeo en 0; quot (-25) 3 == (-8)
  - rem -- Resto de división entera con quot; rem (-25) 3 == (-1)
  - div -- División entera con redondeo en –Inf; div (-25) 3 == (-9)
  - mod -- Resto de división enteracon div; mod (-25) 3 == 2

- Prioridad y asociatividad de operadores binarios con notación infija

```
infixr  9  .
infixr  8  ^, ^^, **
infixl  7  *, /, `quot`,  `rem`,  `div`,  `mod`
infixl  6  +, -
-- The (:) operator is built-in syntax, and cannot legally be given
-- a fixity declaration; but its fixity is given by:
-- infixr  5  :
infix 4  ==,  /=,  <,  <=,  >=,  >
infixr  3  &&
infixr  2  ||
infixl  1  >>,  >>=
infixr  1  =<<
infixr  0  $,  $!,  `seq`
```

2.3 Funciones aritméticas y lógicas básicas

- Funciones aritméticas también incluidas en Prelude
  - **odd** -- Verifica si un número entero es impar
  - **even** -- Verifica si un número entero es par
  - **gcd** -- Calcula el máximo común divisor entre dos enteros
  - **lcm** -- Calcula el mínimo común múltiplo entre dos enteros
  - **subtract** -- Calcula la resta inversa b – a.
  - **abs** -- Calcula el valor absoluto
  - **signum** -- Calcula el signo de un valor numérico (-1, 0, 1)

- Una lista es una colección ordenada de elementos del mismo tipo.
- Haskell permite definir las listas como tipos compuestos a partir de cualquier otro tipo de dato.
- Para describir listas se utilizan corchetes. Por ejemplo, el tipo [*Int*] representa una lista de valores de tipo *Int*.
- En Haskell, las cadenas (*String*) son en realidad listas de caracteres [*Char*].
- Haskell define un valor especial que es la lista vacía, que se denota **[]**.

- Literales de tipo lista:
  - Se pueden definir listas como literales de cualquier tipo separados por coma y entre corchetes. Por ejemplo, **[1, 2, 3, 4]**.
  - Las cadenas son en realidad listas de caracteres. Por ejemplo, **"hola"** es exactamente lo mismo que **['h','o','l','a']**.
  - Se pueden definir listas sobre tipos enumerados utilizando puntos suspensivos. Por ejemplo, **[1 .. 5]** es lo mismo que **[1,2,3,4,5]**.
  - Los números reales también son enumerados utlizando +1.0 para el siguiente. Por ejemplo, **[1.0 .. 2.5]** genera la lista **[1.0, 2.0, 3.0]**.
  - Se puede modificar el incremento utilizando los dos primeros elementos de la lista. Por ejemplo, **[1.0, 1.25 .. 2.0]** genera la lista **[1.0, 1.25, 1.5, 1.75, 2.0]**.
  - De esta forma también se puede crear listas descendentes utilizando incrementos negativos. Por ejemplo, **[5, 4 .. 1]** genera **[5,4,3,2,1]**.

- Operadores sobre listas:
  - (:) : Permite crear una lista con un primer elemento y el resto de la lista. Por ejemplo, **1 : [2,3,4,5]** genera la lista **[1,2,3,4,5]**.
  - (++) : Permite concatenar dos listas. Por ejemplo, **[1,2,3] ++ [4,5]**.
  - (!!) : Obtiene el elemento i-ésimo de la lista. Por ejemplo, **[3..10]!!2 == 5**.
  - En realidad, el tipo de dato lista es una estrudtura con dos campos: el primer elemento y el resto de la lista. Internamente, la lista **[1,2,3,4,5]** se representa como **1:2:3:4:5:[]**.
  - Teniendo en cuenta que Haskell utiliza evaluación perezosa, es bastante frecuente utilizar listas de longitud indefinida en la que solo se evaluan los elementos que resultan necesarios. Por ejemplo, este código describe una función que genera una lista con todos los números naturales a partir de *n*.

```
Prelude> integer n = n : (integer (n+1) )
```

- Funciones básicas sobre listas:
  - **head :: [a] -> a**: Devuelve el primer elemento de una lista.
  - **tail :: [a] -> [a]** : Devuelve el resto de una lista. Genera error sobre la lista vacía.
  - **length :: [a] -> Int** : Devuelve la longitud de la lista.
  - **null :: [a] -> Bool** : Verifica si la lista está vacía.
  - **last :: [a] -> a** : Obtiene el último elemento de la lista.
  - **init :: [a] -> [a]** : Obtiene la lista completa excepto el último elemento.
  - **elem :: a -> [a] -> Bool** : Verifica si un elemento pertenece a una lista.
  - **notElem :: a -> [a] -> Bool** : Verifica si un elemento no pertenece a una lista.

- Funciones sobre cadenas:
  - **lines :: String -> [String]** : Trocea las líneas de una cadena.
  - **unlines :: [String] -> String** : Une las cadenas con el salto de linea.
  - **words :: String -> [String]** : Trocea las palabras de una cadena.
  - **unwords :: [String] -> String** : Une las cadenas con el espacio.

- Funciones sobre listas de booleanos:
  - **and :: [Bool] -> Bool** : Verifica que todos los elementos son True.
  - **or :: [Bool] -> Bool** : Verifica que alguno de los elementos es True.
  - **any :: (a -> Bool) -> [a] -> Bool** : Devuelve True si algún elemento de una lista verifica una función.
  - **all :: (a -> Bool) -> [a] -> Bool** : Devuelve True si todos los elementos de una lista verifican una función.

- Funciones sobre listas de números:
  - **sum :: [Num] -> Num** : Calcula el sumatorio de los elementos.
  - **product :: [Num] -> Num** : Calcula el producto de los elementos.
  - **maximum :: [Ord] -> Ord** : Calcula el máximo de los elementos.
  - **minimum :: [Ord] -> Ord** : Calcula el mínimo de los elementos.

- Funciones que generan listas:
  - **repeat :: a -> [a]** : Genera una lista ilimitada repitiendo el elemento.
  - **replicate :: Int -> a -> [a]** : Genera una lista repitiendo n veces el elemento.
  - **cycle :: [a] -> [a]** : Genera una lista ilimitada repitiendo la lista inicial.
  - **iterate :: (a -> a) -> a -> [a]** : Aplica reiteradamente una función a partir de un valor inicial generando una lista ilimitada.

- Funciones que transforman listas:
  - **map :: (a -> b) -> [a] -> [b]** : Aplica una función a los elementos de una lista.
  - **filter :: (a -> Bool) -> [a] -> [a]** : Selecciona los elementos de una lista que verifican una cierta función.
  - **reverse :: [a] -> [a]** : Devuelve la lista en sentido inverso.

- Funciones que reducen listas:
  - **foldl :: (a -> b -> a) -> a -> [b] -> a** : A partir de un operador binario y un valor inicial, reduce una lista aplicando sucesivamente el operador de izquierda a derecha. (((*a op b1*) *op b2*) *op b3*) ...
  - **foldl1 :: (a -> a -> a) -> [a] -> a** : Es similar a la función *foldl* pero tomando como valor inicial el primer elemento de la lista. ((*b1 op b2*) *op b3*) ...
  - **foldr :: (a -> b -> a) -> a -> [b] -> a** : A partir de un operador binario y un valor inicial, reduce una lista aplicando sucesivamente el operador de derecha a izquierda. (*a op (b1 op (b2 op (... op bn*)
  - **foldr1 :: (a -> a -> a) -> [a] -> a** : Es similar a la función *foldr* pero tomando como valor inicial el primer elemento de la lista. *(b1 op (b2 op (... op bn*)

- Funciones que transforman listas:
  - **scanl :: (a -> b -> a) -> a -> [b] -> [a]** : Es similar a la función *foldl* pero genera uma lista con los valores intermedios. El último valor de la salida de scanl es el mismo valor que devuelve *foldl*.
  - **scanl1 :: (a -> a -> a) -> [a] -> [a]** : Es similar a *foldl1* generando la lista com los valores intermedios.
  - **scanr :: (a -> b -> a) -> a -> [b] -> [a]** : Es similar a la función *foldr* pero genera uma lista con los valores intermedios.
  - **scanr1 :: (a -> a -> a) -> [a] -> [a]** : Es similar a *foldr1* generando la lista com los valores intermedios.

- Funciones que recortan listas:
  - **take :: Int -> [a] -> [a]** : Devuelve los *n* primeros elementos de la lista.
  - **drop :: Int -> [a] -> [a]** : Elimina los *n* primeros elementos de la lista.
  - **takeWhile :: (a -> Bool) -> [a] -> [a]** : Devuelve los primeros elementos de una lista que verifican una función.
  - **dropWhile :: (a -> Bool) -> [a] -> [a]** : Elimina los primeros elementos de una lista que verifican una función.

2.5 Tuplas

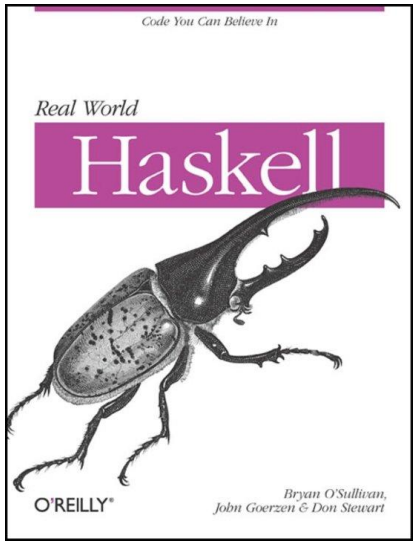
- Una tupla es un tipo de datos formado por una secuencia ordenada de elementos con una estructura y tamaño fijo.
- En Haskell las tuplas se representan entre paréntesis y separadas por coma. Por ejemplo, **(1, True).**
- Una 2-tupla es una pareja de valores **(a,b)**. Una 3-tupla es un trío de valores **(a,b,c)**.

- Funciones sobre tuplas:
  - **fst :: (a,b) -> a** : Obtiene el primer valor de una 2-tupla.
  - **snd:: (a,b) -> a** : Obtiene el segundo valor de una 2-tupla.
  - **splitAt :: Int -> [a] -> ([a], [a])** : Divide una lista en un primer trozo de *n* elementos y un segundo trozo con el resto.
  - **span :: (a -> Bool) -> [a] -> ([a], [a])** : Divide una lista en un trozo con los primeros elementos que verifican una función y el resto de la lista.
  - **break :: (a -> Bool) -> [a] -> ([a], [a])** : Divide una lista en un trozo con los primeros elementos que no verifican una función y el resto de la lista.

- Funciones que empaquetan listas:
  - **zip :: [a] -> [b] -> [(a, b)] :** Toma dos listas y genera una lista emparejando los elementos de ambas. Si las listas no tienen la misma longitud, los elementos que sobran se pierden.
  - **zip3 :: [a] -> [b] -> [c] -> [(a, b, c)]** : Es similar a *zip* pero tomando tres listas y generando tripletas.
  - **zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]** : Es similar a *zip*, pero en vez de generar tuplas aplica la función sobre la pareja y almacena el resultado.
  - **zipWith3 :: (a -> b -> c -> d) -> [a] -> [b] -> [c] -> [d]** : Es similar a *zipWith* pero trabajando sobre tres listas
  - **unzip :: [(a, b)] -> ([a], [b])** : Transforma una lista de parejas en una pareja de listas.
  - **unzip3 :: [(a, b, c)] -> ([a], [b], [c])** : Transforma una lista de tripletas en una tripleta de listas.

Ejercicios:

- Probar en el intérprete de Haskelll los diferentes ejemplos incluidos en el capítulo “Types and Functions” del libro “Real World Haskell”



http://book.realworldhaskell.org/read/types-and-functions.html