Tipos de datos y clases de tipos

Taller de Álgebra I

Segundo cuatrimestre de 2016

Tipos de datos

Tipo de dato

Un **conjunto de valores** a los que se les puede aplicar un **conjunto de funciones**.

Ejemplos

1. $\texttt{Integer} = (\mathbb{Z}, \{+, -, *, \texttt{div}, \texttt{mod}\})$ es el tipo de datos que representa a los enteros con las operaciones aritméticas habituales.
2. $\texttt{Float} = (\mathbb{Q}*, \{+, -, *, /\})$ es el tipo de datos que "representa" a los racionales, con la aritmética de punto flotante.
3. $\texttt{Bool} = (\{\texttt{True}, \texttt{False}\}, \{\&\&, ||, \texttt{not}\})$ representa a los valores lógicos.

Dado un valor de un tipo de datos, solamente se pueden aplicar a ese valor las operaciones definidas para ese tipo de datos.

Tipos de datos

En Haskell los tipos se notan con `::`. Por ejemplo, en GHCI podemos ver el tipo del siguiente valor:

```
Prelude> :t True
True :: Bool
```

A las expresiones también les corresponde un tipo de dato.

```
Prelude> :t (4^10000000 + 2) < 1
(4^10000000 + 2) < 1 :: Bool
```

Dada una expresión, se puede determinar su tipo sin saber su valor.

Tipos de datos

¿Qué tipo tiene la expresión?

```
f True
```

Depende de `f`. Por ejemplo:

```
f :: Bool -> Bool
f x = not x
```

```
f :: Bool -> Float
f x = pi
```

```
funcion3 :: Integer -> Integer -> Bool -> Bool
funcion3 x y b = b || (x > y)
```

Aplicación de funciones

```
Prelude>:t funcion3 10 20 True
funcion3 10 20 True :: Bool
```

Tipos de datos

Ejercicios

Determinar el tipo de las siguientes funciones

```
doble :: ??
doble x = x + x

cuadruple :: ??
cuadruple x = doble (doble x)
```

Ejercicios

Determinar el tipo de las siguientes expresiones

- `doble 10`
- `dist (dist pi 0 pi 1) (doble 0) (doble 2) (3/4)`
- `doble True`

Ejercicios

Ejercicios

Tipar e implementar las siguientes funciones

- `esPar`: dado un valor determina si es par o no. Usar `mod x y`.
- `esPositiva`: dado un valor determina si es positivo

Signatura

- Es importante observar la signatura de las funciones en las definiciones anteriores.
- Especificamos explícitamente el tipo de datos del dominio y el codominio de las funciones que definimos.

1. No es estrictamente necesario especificarlo, dado que el mecanismo de inferencia de tipos de Haskell puede deducir la signatura más general para cada función.
2. Sin embargo, es buena idea dar explícitamente la signatura de las funciones (¿por qué?).

Variables de tipo

A veces las funciones que queremos escribir pueden funcionar sobre muchos tipos de datos. Por ejemplo:

```
identidad :: a -> a
identidad x = x
```

Notar que a va en minúscula y denota una variable de tipo.

En Haskell esa función ya existe y se llama `id` :

```
:t id
id :: a -> a
```

Esta función vale para cualquier tipo de datos.

¿Qué pasa con?

- `id (1>3)`
- `id (sqrt 2)`
- `:t id (1>3)`
- `:t id (sqrt 2)`

Clases de tipos

¿La función `(+)` admite cualquier tipo de datos?

¿Qué pasa con?

- `2 + 4`
- `True + False`

```
:t (+)
(+) :: Num a => a -> a -> a
```

¿ Qué significa `Num a => ...` ?

Lo que aparece antes del símbolo `=>` es la condición que debe cumplir la variable de tipo `a`. La función `(+)` solo se admiten tipo de datos numéricos.

¿Cuál es la signatura de...?

- `:t (>=)`
- `:t (==)`

```
:t (>=)
(>=) :: Ord a => a -> a -> a
```

```
:t (==)
(==) :: Eq a => a -> a -> a
```

Clases de tipos

Clase de tipo

Un **conjunto de tipos de datos** a los que se les puede aplicar un **conjunto de funciones**.

Algunas clases:

1. `Num` := ({ `Int`, `Integer`, `Float`, `Double`, ..., }, { `(+)`, `(*)`, `abs`, ...})
2. `Integral` := ({ `Int`, `Integer`, ..., }, { `mod`, `div`, ...})
3. `Fractional` := ({ `Float`, `Double`, ..., }, { `(/)`, ...})
4. `Floating` := ({ `Float`, `Double`, ..., }, { `sqrt`, `sin`, `cos`, `tan`, ...})
5. `Ord` := {`Bool`, `Int`, `Integer`, `Float`, `Double`, ... }, { `(<=)`, `compare`}
6. `Eq` := ({ `Bool`, `Int`, `Integer`, `Float`, `Double`, ... }, { `(==)`, `(/=)`}

Tipos de datos: Tuplas

- Dados dos tipos de datos `A` y `B`, tenemos el tipo de datos `(A,B)` que representa pares ordenados de elementos, donde el primero es de tipo `A` y el segundo es de tipo `B`.
- Algunas operaciones: `fst` y `snd`
  `fst (1 + 4,2)` $\rightsquigarrow$ `5`
  `snd (1,(2,3))` $\rightsquigarrow$ `(2,3)`
- Ahora podemos definir la distancia un poco más claramente:

```
normaVectorial :: (Float,Float) -> Float
normaVectorial p = sqrt ( (fst p) ^ 2 + (snd p) ^2 )
```

Nota:

- Hay tuplas de distintos tamaños: `(True,1,4.0)`, `(0,0,0,0)`.

Tipos de datos

Ejercicios

- Implementar las siguientes funciones
  - `crearPar :: a -> b -> (a,b)`
  - `invertir :: (a,b) -> (b,a)`
  - `distancia :: (Float,Float) -> (Float,Float) -> Float`
- Completar la implementación de la función `raices :: Float -> Float -> Float -> (Float, Float)` para que esta vez devuelva las dos raíces de la función cuadrática.

Un nuevo tipo: Listas

Tipo Lista

Las listas pueden contener elementos de cualquier tipo (incluso listas).

- `[div 1 1, div 2 1] :: Integral t => [t]`. (Vale `[Integer]`. NO vale `[Float]`).
- `[1, 2] :: Num t => [t]`. (Vale `[Float]`, `[Integer]`).
- `[1.0, 2] :: Fractional t => [t]`. (Vale `[Float]`. NO vale `[Integer]`).
- `[[1], [2,3], [], [1,1000,2,0]] :: Num t => [[t]]`
- `[1, True]` $\rightsquigarrow\rightsquigarrow\rightsquigarrow$ NO ES UNA LISTA VÁLIDA, ¿por qué?
- `[1.0, div 1 1]` $\rightsquigarrow\rightsquigarrow\rightsquigarrow$ NO ES UNA LISTA VÁLIDA, ¿por qué?

Ejercicio

Tipar las siguientes expresiones

- `[(1,2), (3,4), (5,2)]`
- `[maximo 2 3, fst (2+2, 3+4), 3+4 - 3/4]`
- `[ [], [], [], [], [] ]`
- `[]`

Operaciones

Algunas operaciones

- `head :: [a] -> a`
- `tail :: [a] -> [a]`
- `(:) :: a -> [a] -> [a]`
- `(++) :: [a] -> [a] -> [a]`
- `length :: [a] -> Int`
- `reverse :: [a] -> [a]`

Tipar y evaluar las siguientes expresiones

- `head [(1,2), (3,4), (5,2)]`
- `tail [1,2,3,4,4,3,2,1]`
- `head []`
- `head [1,2,3] : [2,3]`
- `[True, True] ++ [False, False]`
- `[1,2] : []`

Listas

Formas rápidas para crear listas

Prueben las siguientes expresiones en GHCI

- `[1..100]`
- `[1,3..100]`
- `[100..1]`

Ejercicios

- Definir la función `listar :: a -> a -> a -> [a]` que toma 3 elementos y los convierte en una lista.
- Escribir una expresión que denote la lista estrictamente decreciente que comienza con el número 1 y termina con el número -100.

Tipos Char y String

Char

Es un tipo de datos cuyos valores representan los caracteres: letras, números, símbolos especiales. Por ejemplo: `'a'`, `'b'`, `'1'`, `'$'`, `'@'`, `' '`, etc.

String

Es una lista de `Char`, es decir `String = [Char]`.
Ejemplo: `"hola"` = `['h', 'o', 'l', 'a']`.

Algunas operaciones

- `isNumber`: devuelve verdadero si el caracter representa un número.
- `isLower`: devuelve verdadero si el caracter es minúscula.
- `toUpper`: dado un caracter, devuelve el mismo en mayúscula.
- `ord`: dado un caracter, devuelve el valor de la enumeración correspondiente a ese caracter.
  Ej: ord 'a' $\rightsquigarrow$ 97
- `chr`: dado un valor devuelve el caracter que representa en la enumeración.
  Ej: chr 97 $\rightsquigarrow$ 'a'

Estas funciones están en un `module` que lo pueden cargar así:

```
Prelude> import Data.Char
Prelude Data.Char>
```

Ejercicios

1. Implementar la función

```
pendiente :: (Float,Float) -> (Float,Float) -> Float
```

que toma dos puntos y calcula la pendiente de la recta que pasa por esos puntos.

2. Implementar la función `iniciales :: String -> String -> String`
que dado el nombre y apellido de una persona, devuelve sus iniciales con puntos. Por ejemplo: `iniciales "Harry" "Potter"` $\rightsquigarrow$ `"H.P."`