Tipos y Clases

Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

Tipos en Haskell (I)

- Un tipo es una colección de valores relacionados.
- Fuertemente tipado: todos los valores/expresiones tienen un tipo
- De tipos seguros: antes de calcular el valor de una expresión comprueba si hay errores de tipo

```
Prelude> :t not
not :: Bool -> Bool

Prelude> head [1..]
1
*Main> head ['a', True]
<interactive>:18:12:
    Couldn't match expected type 'Char' with actual type 'Bool'
    In the expression: True
    In the first argument of 'head', namely '['a', True]'
```

- Los nombres de los tipos **siempre** empiezan con mayúscula

Tipos en Haskell (II)

- Inferencia de tipo: Si no se especifica, Haskell inferirá uno (el más general)
- Predefinidos: `Bool`, `Int` (enteros de precisión fija: entre -2^{31} y $2^{31}-1$), `Integer`, `Float`, `Double` (reales de precisión doble), `Char`, `String`

```
verdadero = True

segundoElemento lista =  head (tail lista)
```

```
*Main> :t verdadero
verdadero :: Bool

*Main> :t segundoElemento
segundoElemento :: [a] -> a
```

Valores lógicos

`Bool`

- Sólo dos elementos: `False` y `True`
- Funciones que los tienen como argumento/resultado

```
Prelude> (True && False) || False
False
Prelude> False && 1

<interactive>:3:10:
    No instance for (Num Bool) arising from the literal ‘1’
    In the second argument of ‘(&&)’, namely ‘1’
    In the expression: False && 1
    In an equation for ‘it’: it = False && 1
Prelude> 30 < 31
True
Prelude> :t True
True :: Bool
```

Definición de constantes

<constante> :: <tipo>

<constante> = ...

```
tres :: Int
tres = 3
```

Si . . . entonces . . . en otro caso . . .

`if <condición> then <consecuencia> else <alternativa>`

- `<consecuencia>` y `<alternativa>` del mismo tipo.
- `<condición>` de tipo `BOOL`

```
clasifica :: Int -> String
clasifica n = if (n > 100) then "Muchos" else "Pocos"
```

```
*Main> clasifica 1500
"Muchos"
*Main> clasifica 90
"Pocos"
```

Listas

`[<tipo>]`

- Entre corchetes
- Los elementos separados por comas (cualquier cantidad)
- Lista vacía: `[]`
- Todos los elementos del mismo tipo

```
*Main> :t [tres]
[tres] :: [Int]
*Main> :t [tres, 4, 5, tres]
[tres, 4, 5, tres] :: [Int]
*Main> [tres, 4, 5, tres]
[3,4,5,3]
*Main> [tres, 4, 5, "tres"]

<interactive>:21:14:
    Couldn't match expected type `Int' with actual type `[Char]'
    In the expression: "tres"
    In the expression: [tres, 4, 5, "tres"]
    In an equation for `it': it = [tres, 4, 5, ....]
```

Enumeración/Concatenación/Constructor

```
*Main> [1..10]
[1,2,3,4,5,6,7,8,9,10]
*Main> [1, 1.25 .. 4]
[1.0,1.25,1.5,1.75,2.0,2.25,2.5,2.75,3.0,3.25,3.5,3.75,4.0]
*Main> [1, 1.25 .. 4.1]
[1.0,1.25,1.5,1.75,2.0,2.25,2.5,2.75,3.0,3.25,3.5,3.75,4.0]
```

```
*Main> [1,2,3] ++ [4,5,6]
[1,2,3,4,5,6]
```

```
*Main> 1:[2,3]
[1,2,3]
```

Cadenas de caracteres

- El tipo es String
- Equivale a [Char]

```
*Main> "Un caso" == ['U', 'n', ' ', 'c', 'a', 's', 'o']
True
*Main> :t "Un caso"
"Un caso" :: [Char]
*Main> :t 'U'
'U' :: Char
*Main> "Un " ++ "caso"
"Un caso"
*Main> 'U' : "n caso"
"Un caso"
```

Tuplas

(<tipo1> {, <tipo2>}+)

- Entre paréntesis
- Los elementos separados por comas (misma cantidad)
- Cada elemento de un tipo (según su posición)

```
*Main> :t (True, "Una")
(True, "Una") :: (Bool, [Char])
```

- Existe el tipo `()` (tiene un único elemento, precisamente `()`)
- No existen las tuplas unitarias: `(<tipo>)`: debe utilizarse el `<tipo>` directamente

Tipos de funciones

Una **función** es una aplicación de valores de un tipo en valores de otro tipo. Así, $T_1 \to T_2$ es el tipo de las funciones que aplican valores del tipo T_1 en valores del tipo T_2.

```
<función> ::  {<tipoArgumento1> -> }+ <tipoRango>
<función> {<parámetro1> }+ = ...
```

```
multiplo3 :: Int -> Bool
multiplo3 n = (n `mod` tres) == 0

promedio3 :: Double -> Double -> Double -> Double
promedio3 x y z = (x + y + z) / 3
```

```
*Main> tres
3
*Main> multiplo3 17
False
*Main> promedio3 3.5 6.78 12
7.426666666666667
```

Parcialización

Las funciones de más de un argumento tienen, en realidad, un argumento y devuelven otra función con un argumento menos.

```
suma :: Int -> Int -> Int
suma n m = n + m

-- Clásico:
-- sucesor :: Int -> Int
-- sucesor m = suma 1 m

-- Con parcialización
sucesor :: Int -> Int
sucesor = suma 1
```

```
*Main> :t suma 3
suma 3 :: Int -> Int
*Main> sucesor 4
5
```

La función `suma` toma un entero `n` y devuelve la función `suma n`.
La función `suma 1` toma un entero `m` y devuelve la suma de `1` y `m`.
Nota: *Este aspecto del tema se puede encontrar más detalle en esta presentación, de la diapositiva 14 a la 17.*

Polimorfismo

- Un tipo es polimórfico ("tiene muchas formas") si contiene una variable de tipo.
- Una función es polimórfica si su tipo es polimórfico. Por ejemplo, la función length es polimórfica:

```
Prelude> :type length
length :: [a] -> Int
Prelude> length [1, 4, 7, 1]
4
Prelude> length ["Lunes", "Martes", "Jueves"]
3
Prelude> length [reverse, tail]
2
```

- Como vemos, para cualquier tipo a, `length` toma una lista de elementos de tipo a y devuelve un entero.
- Decimos que, en este uso anterior, a es una variable de tipos. Las variables de tipos tienen que empezar por minúscula.

Ejemplos de polimorfismo

```
fst :: (a, b) -> a
fst (1,'x')                          1
fst (True,"Hoy")                     True

head :: [a] -> a
head [2,1,4]                         2
head ['b','c']                       'b'

take :: Int -> [a] -> [a]
take 3 [3,5,7,9,4]                   [3,5,7]
take 2 ['l','o','l','a']             "lo"
take 2 "lola"                        "lo"

zip :: [a] -> [b] -> [(a, b)]
zip [3,5] "lo"                       [(3,'l'),(5,'o')]

id :: a -> a
id 3                                 3
id 'x'                               'x'
```

Clases (I)

- Una clase es una colección de tipos junto con ciertas operaciones sobrecargadas llamadas métodos.
- Básicas:
  - `Eq`: comparables por igualdad (`Bool`, `Char`, `String`, `Int`, `Integer`, `Float`, `Double`; listas y tuplas).

```
(==) :: a -> a -> Bool
(/=) :: a -> a -> Bool
```

  - `Ord`: ordenables (ídem)

```
(<), (<=), (>), (>=) :: a -> a -> Bool
min, max             :: a -> a -> a
```

  - `Show`: mostrables (ídem)

```
show :: a -> String
```

Clases (II)

- `Read`: legibles (ídem)

```
read :: String -> a
```

- `Num`: numéricos (`Int`, `Integer`, `Float`, `Double`)

```
(+), (*), (-)       :: a -> a -> a
negate, abs, signum :: a -> a
```

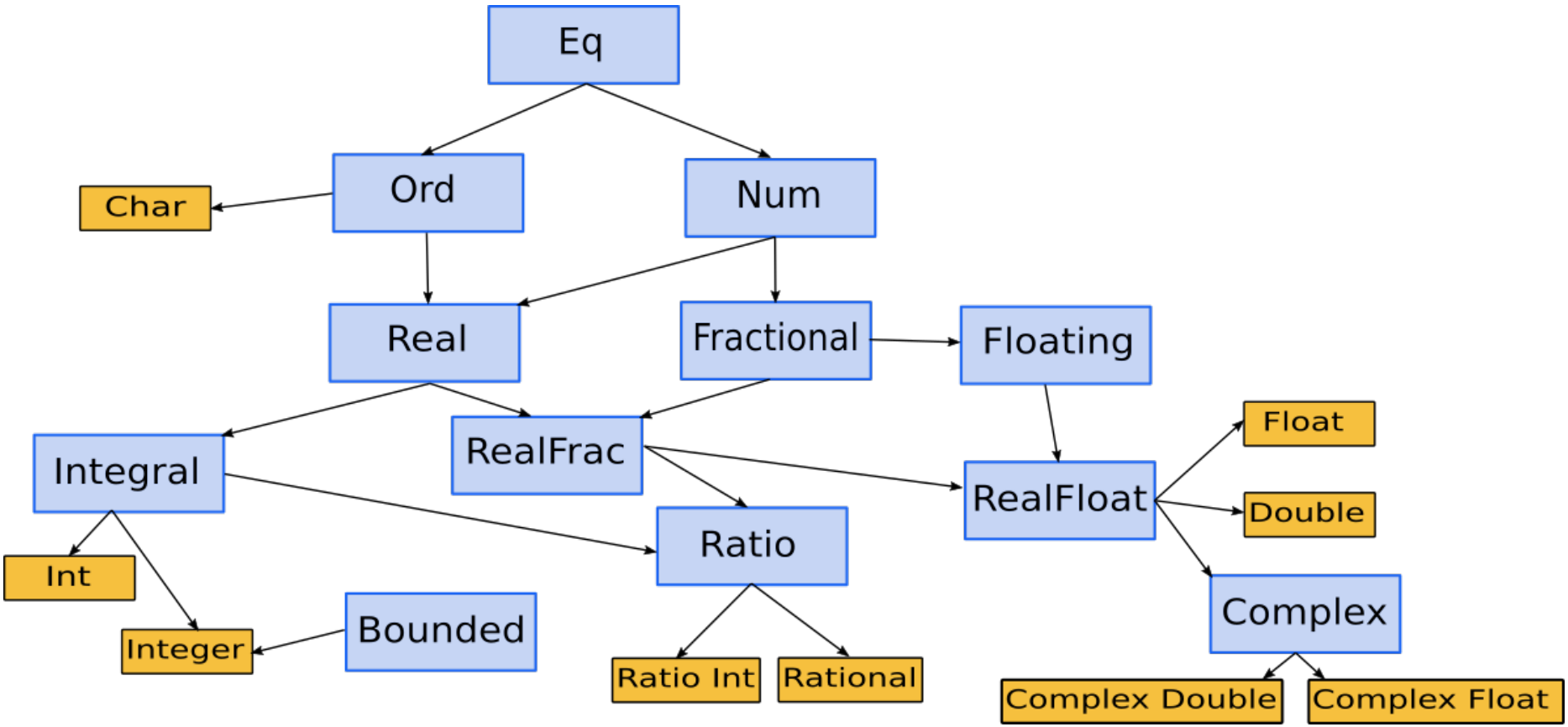
- `Integral`: enteros (`Int`, `Integer`)

```
div :: a -> a -> a
mod :: a -> a -> a
```

`Fractional`: fraccionarios (`Float`, `Double`)

```
(/)   :: a -> a -> a
recip :: a -> a
```

Clases (III)


Eq
Ord
Num
Char
Real
Fractional
Floating
Integral
RealFrac
RealFloat
Float
Double
Int
Ratio
Complex
Integer
Bounded
Ratio Int
Rational
Complex Double
Complex Float

Sobrecarga

- Un tipo está sobrecargado si contiene una restricción de clases.
- Una función está sobrecargada si su tipo está sobrecargado.
- La función `sum` está sobrecargada:

```
Prelude> :type sum
sum :: (Num a) => [a] -> a
Prelude> sum [2, 3, 5]
10
Prelude> sum [2.1, 3.23, 5.345]
10.675
```

- Como vemos, para cualquier tipo **numérico** `a`, `sum` toma una lista de elementos de tipo `a` y devuelve un valor de tipo `a`.
- Decimos que, en este uso anterior, `Num a` es una restricción de clase. Las variables de tipos tienen que empezar por minúscula.
- Las restricciones de clases son expresiones de la forma `C a`, donde `C` es el nombre de una clase y `a` es una variable de tipo.

Ejemplos de sobrecarga

Funciones sobrecargadas:

```
(-) :: (Num a) => a -> a -> a
(*) :: (Num a) => a -> a -> a
negate :: (Num a) => a -> a
abs :: (Num a) => a -> a
signum :: (Num a) => a -> a
```

Números sobrecargados:

```
5 :: (Num t) => t
5.2 :: (Fractional t) => t
```

Preludio

Prelude

```
head :: [a] -> a
Extract the first element of a list, which must be non-empty.
```

Pope B., van Ijzendoorn A. A tour of the Haskell Prelude (basic functions)

```
head
type:
head :: [a] -> a
description:
returns the first element of a non--empty list. If applied to an
empty list an error results.
definition:
head (x:_) = x
usage:
Prelude> head [1..10]
1
Prelude> head ["this", "and", "that"]
"this"
```

Búsqueda

Hoogle (https://www.haskell.org/hoogle/)

Bibliografía I


R. Bird. *Introducción a la programación funcional con Haskell*. Prentice Hall, 2000.
Capítulo 2: Tipos de datos simples


G. Hutton *Programming in Haskell*. Cambridge University Press, 2007.
Chapter 3: Types and classes


B. O'Sullivan, D. Stewart y J. Goerzen. *Real World Haskell*. O'Reilly, 2008.
Chapter 2: Types and Functions


B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. Razonando con Haskell. Thompson, 2004..
Capítulos 2: Introducción a Haskell, y 5: El sistema de clases de Haskell


S. Thompson. Haskell: The Craft of Functional Programming, Second Edition. Addison-Wesley, 1999.
Chapter 3: Basic types and definitions