Programação Funcional
Aula 2 — Tipos e classes

Pedro Vasconcelos
DCC/FCUP

2022

Tipos

Um tipo é um nome para uma coleção de valores relacionados.

Por exemplo, o tipo `Bool` contém os dois valores lógicos:

```
True        False
```

Erros de tipos

Algumas operações só fazem sentido com valores de determinados tipos.

Exemplo: não faz sentido somar números e valores lógicos.

```
> 1 + False
<interactive>:1:1: error:
    • No instance for (Num Bool) arising from a use of ‘+’
    • In the expression: 1 + False
      In an equation for ‘it’: it = 1 + False
```

Em Haskell, estes erros são detetados classificando as expressões com o tipo do resultado.

Tipos em Haskell

Escrevemos

```
e :: T
```

para indicar que a expressão *e* admite o tipo *T*.

- Se *e* :: *T*, então o resultado de *e* será um valor de tipo *T*
- O interpretador/compilador verifica tipos indicados pelo programador e infere os tipos omitidos
- Os erros de tipos são assinalados antes da execução

Tipos básicos

Bool valores lógicos
`True`, `False`

Char carateres simples
`'A'`, `'B'`, `'?'`, `'\n'`

String sequências de carateres
`"Abacate"`, `"UB40"`

Int inteiros de precisão fixa (tipicamente 64-*bits*)
`142`, `-1233456`

Integer inteiros de precisão arbitrária
(apenas limitados pela memória do computador)

Float vírgula flutuante de precisão simples
`3.14154`, `-1.23e10`

Double vírgula flutuante de precisão dupla

Listas

Uma *lista* é uma sequência de tamanho variável de elementos dum mesmo tipo.

```
[False,True,False] :: [Bool]
['a', 'b', 'c', 'd'] :: [Char]
```

Em geral: `[`*T*`]` é o tipo de listas cujos elementos são de tipo *T*.

Caso particular: `String` é equivalente a `[Char]`.

```
"abcd" == ['a','b','c','d']
```

Tuplos

Um *tuplo* é uma sequência de tamanho fixo de elementos de tipos possivelmente diferentes.

```
(42,'a') :: (Int,Char)
(False,'b',True) :: (Bool,Char,Bool)
```

Em geral: $(T_1, T_2, \ldots, T_n)$ é o tipo de tuplos com n componentes de tipos T_i para i de 1 a n.

Observações

- Listas de tamanhos diferentes podem ter o mesmo tipo.
- Tuplos de tamanhos diferentes têm tipos diferentes.

```
['a'] :: [Char]

['b','a','b'] :: [Char]

('a','b') :: (Char,Char)

('b','a','b') :: (Char,Char,Char)
```

Observações (cont.)

Os elementos de listas e tuplos podem ser quaisquer valores, inclusivé outras listas e tuplos.

```
[['a'], ['b','c']] :: [[Char]]

(1,('a',2)) :: (Int,(Char,Int))

(1, ['a','b']) :: (Int,[Char])
```

Observações (cont.)

- A lista vazia `[]` admite qualquer tipo `[`*T*`]`
- O tuplo vazio `()` é o único valor do *tipo unitário* `()`
- Não existem tuplos com apenas um elemento

Tipos funcionais

Uma função faz corresponder valores de um tipo em valores de outro um tipo.

```
not :: Bool -> Bool
```

```
isDigit :: Char -> Bool
```

Em geral:

$$A \texttt{ -> } B$$

é o tipo das funções que fazem corresponder valores do tipo A em valores do tipo B.

Tipos funcionais (cont.)

Os argumento e resultado duma função podem ser listas, tuplos ou de quaisquer outros tipos.

```
soma :: (Int,Int) -> Int
soma (x,y) = x+y

contar :: Int -> [Int]
contar n = [0..n]
```

Funções de vários argumentos

Uma função de vários argumentos toma um argumento de cada vez.

```
soma :: Int -> (Int -> Int)
soma x y = x+y

incr :: Int -> Int
incr = soma 1
```

Ou seja: `soma 1` é a função que a cada y associa $1 + y$.

NB: a esta forma de tratar múltiplos argumentos chama-se *currying* (em homenagem a Haskell B. Curry).

Tuplos vs. *currying*

Função de dois argumentos (*curried*)

```
soma :: Int -> (Int -> Int)
soma x y = x+y
```

Função de um argumento (par de inteiros)

```
soma' :: (Int,Int) -> Int
soma' (x,y) = x+y
```

Porquê usar *currying*?

Funções *curried* são mais flexíveis do que funções usando tuplos porque podemos aplicá-las parcialmente.

Exemplos

```
soma 1 :: Int -> Int                    -- incrementar

take 5 :: [Char] -> [Char]              -- primeiros 5 elms.

drop 5 :: [Char] -> [Char]              -- retirar 5 elms.
```

É preferível usar *currying* exceto quando queremos explicitamente construir tuplos.

Convenções sintáticas

Duas convenções que reduzem a necessidade de parêntesis:

- a seta `->` associa à direita
- a aplicação associa à esquerda

```
   Int -> Int -> Int -> Int
=  Int -> (Int -> (Int -> Int))

  f x y z
= (((f x) y) z)
```

Funções polimorfas

Certas funções operam com valores de qualquer tipo; tais funções admitem tipos com variáveis.

Uma função diz-se polimorfa ("de muitas formas") se admite um tipo com variáveis.

Exemplo

```
length :: [a] -> Int
```

A função *length* calcula o comprimento duma lista de valores de qualquer tipo *a*.

Funções polimorfas (cont.)

Ao aplicar funções polimorfas, as variáveis de tipos são automaticamente substituidas pelos tipos concretos:

```
> length [1,2,3,4]                          -- Int
4
> length "abacate"                          -- Char
7
> length [False,True]                       -- Bool
2
> length [(2,'A'),(3,'C')]                  -- (Int,Char)
2
```

As variáveis de tipo devem começar por uma letra minúscula; é convencional usar *a*, *b*, *c*, ...

Funções polimorfas (cont.)

Muitas funções do prelúdio-padrão são poliformas:

```
null :: [a] -> Bool

head :: [a] -> a

take :: Int -> [a] -> [a]

fst :: (a,b) -> a

zip :: [a] -> [b] -> [(a,b)]
```

O polimorfismo permite utilizar estas funções em vários contextos.

Sobrecarga (*overloading*)

Certas funções operam sobre vários tipos mas não sobre *quaisquer* tipos.

```
> sum [1,2,3]
6

> sum [1.5, 0.5, 2.5]
4.5

> sum ['a', 'b', 'c']
  No instance for (Num Char) ...

> sum [True, False]
  No instance for (Num Bool) ...
```

Sobrecarga (*overloading*) (cont.)

O tipo mais geral da função `sum` é:

```
sum :: Num a => [a] -> a
```

- "`Num a => ...`" é uma restrição da variável *a*.
- Indica que `sum` opera apenas sobre tipos numéricos
- Exemplos: `Int`, `Integer`, `Float`, `Double`

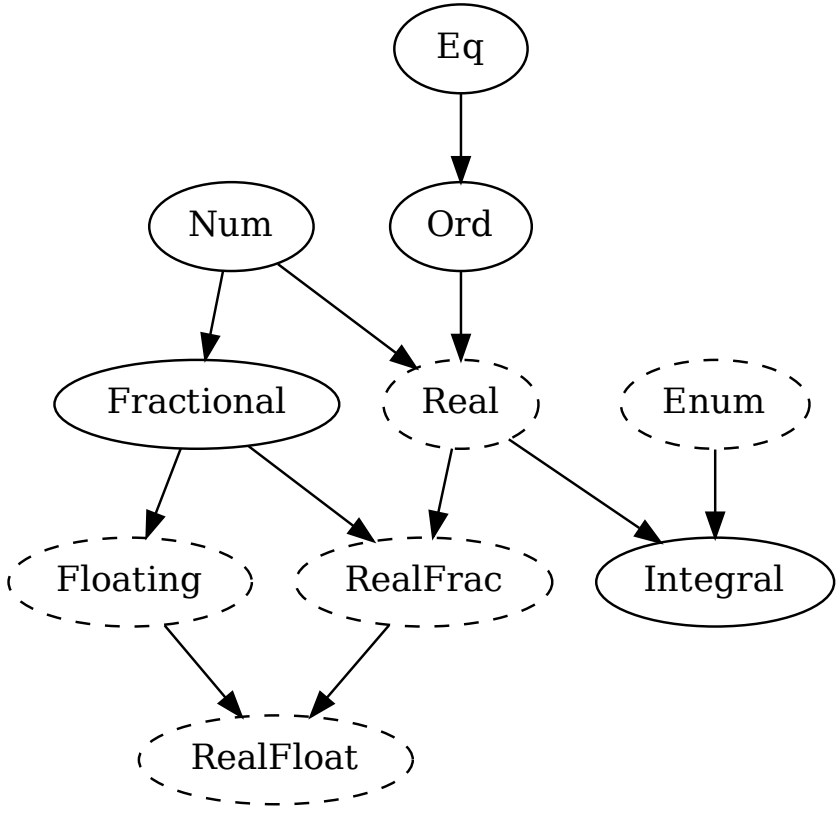
Algumas classes pré-definidas

Num	tipos numéricos (ex: Int, Integer, Float, Double)
Integral	tipos com divisão inteira (ex: Int, Integer)
Fractional	tipos com divisão fracionária (ex: Float, Double)
Eq	tipos com igualdade
Ord	tipos com comparações de ordem total

Exemplos:

```
(+) :: Num a => a -> a -> a
mod :: Integral a => a -> a -> a
(/) :: Fractional a => a -> a -> a
(==) :: Eq a => a -> a -> Bool
(<) :: Ord a => a -> a -> Bool
max :: Ord a => a -> a -> a
```

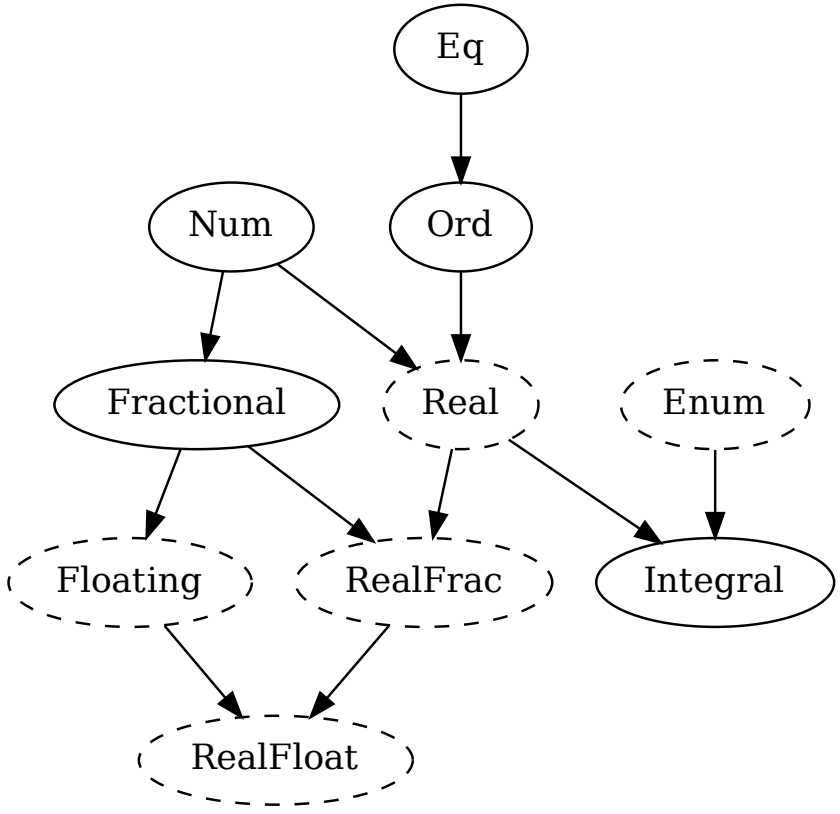
Hierarquia de classes



Algumas classes estão organizadas numa hierarquica:

- Ord é uma subclasse de Eq
- Fractional e Integral são subclasses de Num

Hierarquia de classes (cont.)



B é subclasse de *A* sse todas as operações de *A* existem para tipos em *B*. Exemplos:

- `==` está definida para tipos em Ord
- `+`, `-` e `*` estão definidas para Fractional e Integral

Constantes numéricas

Em Haskell, as constantes também podem ser usadas com vários tipos:

```
1 :: Int
1 :: Float
1 :: Num a => a                    -- tipo mais geral

3.0 :: Float
3.0 :: Double
3.0 :: Fractional a => a           -- tipo mais geral
```

Logo, as expressões seguintes são correctamente tipadas:

```
1/3 :: Float
(1 + 1.5 + 2) :: Float
```

Misturar tipos numéricos

Vamos tentar definir uma função para calcular a média aritmética duma lista de números.

```
media xs = sum xs / length xs
```

Parece correta, mas tem um erro de tipos!

```
Could not deduce (Fractional Int) ...
```

Misturar tipos numéricos (cont.)

Problema

```
(/) :: Fractional a => a -> a -> a  -- divisão fracionária
length xs :: Int                    -- não é fracionário
```

Solução: usar uma conversão explícita

```
media xs = sum xs / fromIntegral (length xs)
```

`fromIntegral` converte qualquer tipo inteiro para qualquer outro tipo numérico.

Quando usar anotações de tipos

- Podemos escrever definições e deixar o interpretador inferir os tipos.
- Mas é recomendado anotar sempre tipos de definições de funções:

```
media :: [Float] -> Float
media xs = sum xs / fromIntegral(length xs)
```

- Benefícios:
  - serve de documentação
  - ajuda a escrever as definições
  - por vezes ajuda a tornar as mensagens de erros mais compreensíveis

Quando usar anotações de tipos (cont.)

- Pode ser mais fácil começar com um tipo concreto e depois generalizar
- O interpretador assinala um erro de tipos se a generalização for errada

```
media :: Num a => [a] -> a          -- ERRO
media xs = sum xs / fromIntegral(length xs)

media :: Fractional a => [a] -> a   -- OK
media xs = sum xs / fromIntegral(length xs)
```