

Programação Funcional

2ª Aula — Tipos e classes

Sandra Alves
DCC/FCUP

2019/20

Tipos

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

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

`True`

`False`

Erros de tipos

Algumas operações só fazem sentido com valores de determinados tipos.

Por exemplo: não faz sentido somar números e valores lógicos.

```
> 1 + False
```

```
ERRO
```

Em Haskell, estes erros são detetados classificando as expressões com os **tipos** dos resultados.

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 **verifica** tipos indicados pelo programador e **infere** tipos omitidos.
- Os programas com erros de tipos são **rejeitados** antes da execução.

Tipos básicos

Bool valores lógicos

True, False

Char caracteres simples

'A', 'B', '?', '\n'

String sequências de caracteres

"Abba", "UB40"

Int inteiros de precisão fixa (32 ou 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 .

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, \dots, T_n) é o tipo de tuplos com n componentes de tipos T_i para i de 1 a n .

Observações I

- 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 II

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])`

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

Tipos funcionais I

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

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

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

Em geral: $T_1 \rightarrow T_2$ é o tipo das funções que fazem corresponder valores do tipo T_1 em valores do tipo T_2 .

Tipos funcionais II

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 \rightarrow associa à **direita**;
- a aplicação associa à **esquerda**.

$$\begin{aligned} & \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} &= \\ = & \text{Int} \rightarrow (\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})) \end{aligned}$$

$$f \ x \ y \ z \quad = \quad (((f \ x) \ y) \ z)$$

Funções polimorfas I

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

$$\text{length} :: [a] \rightarrow \text{Int}$$

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

Funções polimorfos II

Ao aplicar funções polimorfos, as variáveis de tipos são automaticamente substituídas pelos tipos concretos:

```
> length [1,2,3,4]                                -- Int
4
```

```
> length [False,True]                             -- Bool
2
```

```
> length [(0,'X'),(1,'0')]                         -- (Int,Char)
2
```

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

Funções polimorfias III

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 usar estas funções em contextos muito diferentes.

Sobrecarga (*overloading*) I

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']
```

```
ERRO
```

```
> sum [True, False]
```

```
ERRO
```

Sobrecarga (*overloading*) II

Nestes casos o tipo mais geral da função tem *restrições de classe*.

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

- “Num a => ...” é uma **restrição de classe** da variável *a*.
- Indica que sum opera apenas sobre tipos *a* que sejam **numéricos**.

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 *ordem total*

Exemplos

```
(+) :: Num 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
```

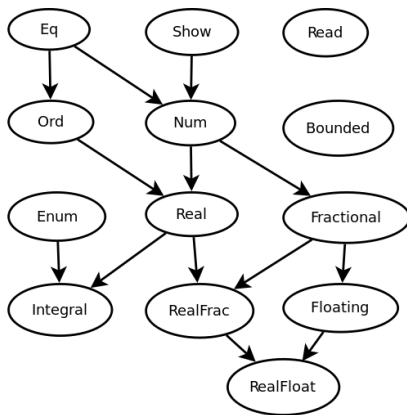
Hierarquia de classes

Algumas classes respeitam uma hierarquica:

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

Assim, podemos usar:

- **=** e **/=** com tipos em **Ord** ou em **Num**
- **+**, **-** e ***** com tipos em **Fractional** ou em **Integral**



Constantes numéricas

Em Haskell, também as constantes numéricas 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
```

Assim, as expressões seguintes são correctamente tipadas:

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

Misturar tipos numéricos

Uma função para calcular a média duma lista de números.

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

Erro de tipos!

```
Couldn't match expected type 'Float' with actual type 'Int'
  In the return type of a call of 'length'
  In the second argument of '(/)', namely 'length xs'
  In the expression: sum xs / length xs
```


Misturar tipos numéricos (cont.)

Problema

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

Solução: usar uma conversão explícita

```
media :: [Float] -> Float  
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.
- É melhor prática **anotar o tipo de cada definição**:
 - serve de documentação;
 - ajuda a escrever as definições;
 - permite mensagens de erro de tipos mais compreensíveis.
- Pode ser mais fácil começar com um tipo concreto e depois generalizar.
- O interpretador dá um erro de tipos se a generalização for errada.
- O tipo mais geral de funções com operações numéricas, igualdade ou comparações, necessita sempre de **restrições de classes**.

Exercícios:

1. Indique os tipos dos seguintes valores:

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

`('a', 'b', 'c')`

`[(False,1), (True,0)]`

`([False,True], ['0', '1'])`

2. Indique os tipos das seguintes funções:

`swap (x,y) = (y,x)`

`pair x y = (x,y)`

`twice f x = f (f x)`

`zip' xs = zip xs xs`