Programação Funcional
Definição de Funções

2021

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 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])
```

Observações III

- 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 I

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 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 `->` 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 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

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

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

Funções polimorfas II

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 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 utilizar estas funções em vários contextos.

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']
  No instance for (Num Char) ...

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

Sobrecarga (*overloading*) II

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 I

Algumas classes estão organizadas numa hierarquica:

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

Hierarquia de classes II

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 I

- 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 II

- 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)
```

Definição de funções

Podemos definir novas funções simples como expressões usando outras funções pré-definidas.

```
minuscula :: Char -> Bool
minuscula c = c>='a' && c<='z'

factorial :: Integer -> Integer
factorial n = product [1..n]
```

Expressões condicionais

Podemos exprimir uma condição com duas alternativas usando 'if...then...else'.

```
-- valor absoluto: x se x>=0; -x se x<0
absoluto :: Float -> Float
absoluto x = if x>=0 then x else -x
```

As expressões condicionais podem ser imbricadas:

```
sinal :: Int -> Int
sinal x = if x>0 then 1 else
            (if x==0 then 0 else -1)
```

Ao contrário do C ou Java:

- o 'if...then ...else' é uma expressão e não um comando
- a alternativa 'else' é obrigatória

Alternativas com guardas I

Podemos usar guardas em vez de expressões condicionais:

```
absoluto :: Float -> Float
absoluto | x>=0      = x
         | otherwise = -x


sinal :: Int -> Int
sinal x | x>0       = 1
        | x==0      = 0
        | otherwise = -1
```

Alternativas com guardas II

Caso geral:

```
f x y ... | condição 1 = expressão 1
          | condição 2 = expressão 2
            ...
          | condição N = expressão N
```

- As condições são testada pela ordem indicada
- O resultado é dado pela expressão da primeira alternativa verdadeira
- A função é indefinida se nenhuma condição for verdadeira (dá erro durante a execução)
- A condição 'otherwise' é um sinónimo para `True`

Alternativas com guardas III

Definições locais abrangem todas as alternativas se a palavra 'where' estiver alinhada com as guardas.

Exemplo: raizes de uma equação do 2º grau.

```
raizes :: Float -> Float -> Float -> [Float]
raizes a b c
     | delta>0   = [(-b+sqrt delta)/(2*a),
                    (-b-sqrt delta)/(2*a)]
     | delta==0  = [-b/(2*a)]
     | otherwise = []
     where delta = b^2 - 4*a*c
```

Alternativas com guardas IV

Também podemos definir variáveis locais usando 'let...in...'.
Neste caso o âmbito da definição não inclui outras alternativas.

```
raizes :: Float -> Float -> Float -> [Float]
raizes a b c
     | delta>0   = let r = sqrt delta
                   in [(-b+r)/(2*a),(-b-r)/(2*a)]
        -- r só está definido na expressão acima
     | delta==0  = [-b/(2*a)]
     | otherwise = []
     where delta = b^2 - 4*a*c
```

Encaixe de padrões I

Podemos usar várias equações com padrões para distinguir casos.

```
not :: Bool -> Bool
not True = False
not False = True

(&&) :: Bool -> Bool -> Bool
True  && True  = True
True  && False = False
False && True  = False
False && False = False
```

Nota: estas funções estão pré-definidas no prelúdio-padrão.

Encaixe de padrões II

Uma definição alternativa:

```
(&&) :: Bool -> Bool -> Bool
False && _ = False
True  && x = x
```

- O padrão "_" encaixa qualquer valor
- As variáveis no padrão podem ser usadas no lado direito
- A definição acima ignora o segundo argumento se o primeiro for `False`

Encaixe de padrões III

Não podemos repetir variáveis nos padrões:

```
x && x = x                    -- ERRO
_ && _ = False
```

Alternativa: podemos usar uma guarda para impor a condição de igualdade.

```
x && y | x==y = x             -- OK
_ && _        = False
```

Padrões sobre tuplos

Exemplos: as funções `fst` (*first*) e `snd` (*second*) dão-nos o primeiro e segundo elemento de um par.

```
fst :: (a,b) -> a
fst (x,_) = x

snd :: (a,b) -> b
snd (_,y) = y
```

Estas funções também estão pré-definidas no prelúdio-padrão.

Construtor de listas

Qualquer lista é construida acrescentando elementos um-a-um a uma lista vazia usando "`:`"[1].

```
[1, 2, 3, 4]   =   1 : (2 : (3 : (4 : [])))
```

[1] Lê-se "*cons*" de "*construtor*".

Padrões sobre listas I

Podemos também usar um padrão `x:xs` para decompor uma lista.

```
head :: [a] -> a
head (x:_) = x                    -- 1º elemento

tail :: [a] -> [a]
tail (_:xs) = xs                  -- restantes elementos
```

Padrões sobre listas II

O padrão `x:xs` só encaixa listas não-vazias:

```
> head []
```
ERRO

Necessitamos de parêntesis à volta do padrão (porque a aplicação têm maior precedência):

```
head x:_ = x                    -- ERRO

head (x:_) = x                  -- OK
```

Padrões sobre inteiros I

Exemplo: testar se um inteiro é 0, 1 ou 2.

```
small :: Int -> Bool
small 0    = True
small 1    = True
small 2    = True
small _    = False
```

A última equação encaixa todos os casos restantes.

Expressões-case I

Em vez de equações podemos usar ‘case. . . of. . . ’:

Exemplo:

```
null :: [a] -> Bool
null xs = case xs of
              [] -> True
              (_:_) -> False
```

Expressões-case II

Os padrões são tentados pela ordem das alternativas.

Logo, a esta definição é equivalente à anterior:

```
null :: [a] -> Bool
null xs = case xs of
            [] -> True
            _  -> False
```

Expressões-lambda I

Podemos definir uma *função anónima* (i.e. sem nome) usando uma expressão-lambda.

Exemplo:

```
\x -> 2*x+1
```

é a função que a cada x faz corresponder $2x + 1$.

Esta notação é baseada no *cálculo-*λ, o formalismo matemático que é a base teórica da programação funcional.

Expressões-lambda II

Podemos usar uma expressão-lambda aplicando-a a um valor (tal como o nome de uma função).

```
> (\x -> 2*x+1) 1
3

> (\x -> 2*x+1) 3
7
```

Para que servem as expressões-lambda? I

As expressões-lambda permitem definir funções cujos resultados são outras funções.

Em particular: as expressões-lambda permitem compreender o uso de *"currying"* para funções de múltiplos argumentos.

Exemplo:

```
soma x y = x+y
```

é equivalente a

```
soma = \x -> (\y -> x+y)
```

Para que servem as expressões-lambda? II

As expressões-lambda são também úteis para evitar dar nomes a funções curtas.

Um exemplo usando `map` (que aplica uma função a todos os elementos duma lista): em vez de

```
quadrados = map f [1..10]
      where f x = x^2
```

podemos escrever

```
quadrados = map (\x->x^2) [1..10]
```

Operadores e secções I

Qualquer operador binário (+, ∗, etc.) pode ser usado como função de dois argumentos colocando-o entre parentêsis.

Exemplo:

```
> 1+2
3

> (+) 1 2
3

> (++) "Abra" "cadabra!"
"Abracadabra!"
```

Operadores e secções II

- Expressões da forma `(x ⊗)` e `(⊗ x)` chamam-se secções
- Definem a função resultante de aplicar um dos argumentos do operador ⊗

```
> (+1) 2
3

> (/2) 1
0.5

> (++"!!!") "Bang"
"Bang!!!"
```

Exemplos

`(1+)`	sucessor
`(2*)`	dobro
`(^2)`	quadrado
`(1/)`	recíproco
`(++"!!")`	concatenar `"!!"` ao final

Assim podemos re-escrever o exemplo

```
quadrados = map (\x -> x^2) [1..10]
```

de forma ainda mais sucinta:

```
quadrados = map (^2) [1..10]
```