

Tipos e Classes

MCTA016-13 - Paradigmas de Programação

Emilio Francesquini, Mario Leston Rey
{e.francesquini, mario.leston}@ufabc.edu.br

2023.Q2

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC



- Estes slides foram preparados para o curso de **Paradigmas de Programação na UFABC**.
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.
- Conteúdo baseado no texto preparado, e gentilmente cedido, pelo Professor Fabrício Olivetti de França da UFABC.



Tipos e Classes Padrões

- Um **tipo** é uma coleção de valores relacionados entre si.

Exemplos

- **Int** compreende todos os valores de números inteiros.
- **Bool** contém apenas os valores **True** e **False**, representando valores lógicos

- Em Haskell, os tipos são definidos pela notação

1 `v :: T`

- Significando que `v` define um valor do tipo `T`.

```
1 False :: Bool
2 True  :: Bool
3 10    :: Int
```

- De forma similar uma função pode ser definida por

1 $f :: T0 \rightarrow T1$

- Indicando que a função f recebe um valor do tipo $T0$ e retorna um valor do tipo $T1$.

- O tipo da aplicação de uma função é o tipo do seu retorno:

```
1 False :: Bool
2 not  :: Bool -> Bool
3 not False :: Bool
```

- Em Haskell, **toda** expressão tem um tipo calculado antes de avaliar o resultado da expressão.
- Os tipos podem ser definidos automaticamente pela inferência do tipo.

Por exemplo, se eu tenho:

1 $f :: A \rightarrow B$

2 $e :: A$

então

1 $f e :: B$

```
1 impar x = x `rem` 2 == 1
```

Pergunta

Qual o tipo da função?

Abra o `ghci` e digite:

```
1 :t (`rem` 2)
```

Abra o `ghci` e digite:

```
1 :t (`rem` 2)
2 (`rem` 2) :: Integral a => a -> a
```

Logo `x` deve ser do tipo `Integral` e a função deve ser:

```
1 impar :: Integral a => a -> ???  
2 impar x = x `rem` 2 == 1
```

```
1  :t (== 1)
2  (== 1) :: (Eq a, Num a) => a -> Bool
```

Isso restringe ainda mais nosso tipo, como veremos mais a frente. Por ora, observemos `a -> Bool`.

A assinatura da função fica:

```
1 impar :: (Eq a, Integral a) => a -> Bool
2 impar x = x `rem` 2 == 1
```

Se eu fizer (ou tentar):

```
1 r1 = impar "3"
```

Isso vai gerar um **erro de compilação!**

- No instance for (Integral [Char]) arising from a use of 'impar'
- In the expression: `impar "3"`
In an equation for 'r1': `r1 = impar "3"`

Tipos Básicos

- O compilador GHC já vem com suporte nativo a diversos tipos básicos.
- Durante o curso veremos como definir e criar os nossos próprios tipos.

Os tipos são:

- **Bool**: contém os valores **True** e **False**. Expressões booleanas podem ser construídas com os operadores **&&** (e), **||** (ou) e **not**.
- **Char**: contém todos os caracteres no sistema **Unicode**. Podemos representar a letra 'a', o número '5', a seta tripla '≡' e o *homem de terno levitando*¹ '🙈'.
- **String**: sequências de caracteres delimitados por aspas duplas: "Olá Mundo".

¹Este é o nome oficial do caracter na tabela Unicode v.7.0!

- **Int**: inteiros com precisão fixa em 64 bits. Representa os valores numéricos de -2^{63} até $2^{63} - 1$.
- **Integer**: inteiros de precisão arbitrária. Representa valores inteiros de qualquer precisão, a memória é o limite. Mais lento do que operações com **Int**.
- **Float**: valores em ponto-flutuante de precisão simples. Permite representar números com um total de 7 dígitos, em média.
- **Double**: valores em ponto-flutuante de precisão dupla. Permite representar números com quase 16 dígitos, em média.

Note que ao escrever:

```
1 x = 3
```

O tipo de `x` pode ser `Int`, `Integer`, `Float` ou `Double`.

Pergunta

Qual tipo devemos atribuir a `x`?

Listas são sequências de elementos do mesmo tipo agrupados por colchetes e separados por vírgula:

1 [1, 2, 3, 4]

Uma lista de tipo T tem tipo [T]:

```
1 [1,2,3,4] :: [Int]
2 [False, True, True] :: [Bool]
3 ['o', 'l', 'a'] :: [Char]
```

- O **tamanho** da lista (*length*) representa a quantidade de elementos que ela contém.
- Uma lista vazia é representada por []
- Listas com um elemento, como [1], [False] e [[]] são chamadas **singleton**.

- Como podem ter percebido no slide anterior, podemos ter listas de listas:

```
1 [ [1,2,3], [4,5] ] :: [[Int]]
2 [ [ 'o', 'l', 'a' ], [ 'm', 'u', 'n', 'd', 'o' ] ] :: [[Char]]
```

Notem que:

- O tipo da lista não especifica seu tamanho
- Não existe limitação quanto ao tipo da lista
- Não existe limitação quanto ao tamanho da lista

- **Tuplas** são sequências finitas de componentes, contendo zero ou mais tipos diferentes:

```
1 (True, False) :: (Bool, Bool)
2 (1.0, "Sim", False) :: (Double, String, Bool)
```

- O tipo da tupla é definido como (T_1, T_2, \dots, T_n) .

- O número de componentes de uma tupla é chamado **aridade** (*arity*)
- Uma tupla de aridade zero, a tupla vazia, é representada por ()
- Tuplas de tamanho dois são conhecidas como *duplas*, já as de tamanho três são *triplas*.

Notem que:

- O tipo da tupla especifica seu tamanho
- Não existe limitações aos tipos associados à tupla (podemos ter tuplas de tuplas)
- Tuplas **devem** ter um tamanho finito (e fixo!)
- Tuplas de aridade 1 não são permitidas para manter compatibilidade do uso de parênteses para definir a ordem de avaliação

- **Funções** são mapeamentos que transformam um valor de um tipo de entrada, digamos **T1** em um valor de um tipo de saída, **T2**. O tipo (também conhecido como assinatura) de uma função é escrito como **T1 -> T2**.

1 `not :: Bool -> Bool`

2 `even :: Int -> Bool`

- Como não existem restrições para os tipos, a noção de mapa de um tipo para outro é suficiente para escrever funções com 0 ou mais argumentos e que retornem 0 ou mais valores.

Exercício

Crie as seguintes funções em um arquivo `aula02.hs`, carregue no `ghci`, verifique seu tipo e teste com algumas entradas:

```
1 soma :: (Int, Int) -> Int
2 soma (x,y) = x+y
3
4 zeroAteN :: Int -> [Int]
5 zeroAteN n = [0..n]
```

- Uma função pode ser **total** se ela for definida para qualquer valor do tipo de entrada ou **parcial** se existem algumas entradas para qual ela não tem valor de saída definido:

```
1 > head []  
2 *** Exception: Prelude.head: empty list
```





- É óbvio que o pessoal do Unicode também criou um símbolo para o curry

japonês (karê, カレー): .

```
1 Prelude> :t '🍛'  
2 '🍛' :: Char  
3 Prelude>
```

- Funções com múltiplos argumentos podem ser definidas de uma outra forma, inicialmente não óbvia, mas que torna sua representação mais natural.

Como não existem restrições de tipos, uma função pode retornar uma outra função:

```
1 soma' :: Int -> (Int -> Int)
2 soma' x = \y -> x + y
```

- Ela recebe um valor x e retorna uma função que recebe um y e devolve $y + x$ (aprenderemos sobre $\backslash y$ mais adiante).

```
1 soma' :: Int -> (Int -> Int)
2 soma' x = \y -> x + y
```

A seguinte definição ainda é válida:

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

Ela indica que a função `soma'` recebe um valor `x`, cria uma função `\y -> x + y` e aplica com o valor `y`. Isso é conhecido como **curried functions**.

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

Da mesma forma podemos ter:

```
1 mult :: Int -> (Int -> (Int -> Int))  
2 mult x y z = x*y*z
```

Para evitar escrever um monte de parênteses (como no Lisp 😊), a seguinte sintaxe é válida:

```
1 soma' :: Int -> Int -> Int
2 soma' x y = x + y
3
4 mult :: Int -> Int -> Int -> Int
5 mult x y z = x*y*z
```

Polimorfismo

Considere a função `length` que retorna o tamanho de uma lista. Ela deve funcionar para qualquer uma dessas listas:

```
1 [1,2,3,4] :: [Int]
2 [False, True, True] :: [Bool]
3 ['o', 'l', 'a'] :: [Char]
```

Pergunta

Qual é então o tipo de `length`?

- Qual o tipo de `length`?

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

- Quem é `a`?

- Em Haskell, `a` é conhecida como **variável de tipo** e ela indica que a função deve funcionar para listas de qualquer tipo.
- As variáveis de tipo devem seguir a mesma convenção de nomes do Haskell, iniciar com letra minúscula. Como convenção utilizamos `a`, `b`, `c`, `...`

- Considere agora a função (+), diferentemente de `length` ela pode ter comportamentos distintos para tipos diferentes.
- Internamente, a adição de dois valores do tipo `Int` é, em geral, diferente da adição de dois valores do tipo `Integer` (e, certamente, é diferente da adição de dois valores do tipo `Float`).
- Ainda assim, essa função **deve** ser aplicada a tipos numéricos.

- A ideia de que uma função pode ser aplicada a apenas uma classe de tipos é explicitada pela **Restrição de classe** (*class constraint*).
- Uma restrição é escrita na forma $C \ a$, onde C é o nome da classe e a uma variável de tipo.

1 $(+) \ :: \ \mathbf{Num} \ a \ \Rightarrow \ a \ \rightarrow \ a \ \rightarrow \ a$

- A função $+$ recebe dois tipos de uma classe numérica e retorna um valor desse mesmo tipo.

- Note que nesse caso, ao especificar a entrada como **Int** para o primeiro argumento, todos os outros **devem** ser **Int** também.

1 (+) :: Num a => a -> a -> a

- Uma vez que uma função contém uma restrição de classe, pode ser necessário definir **instâncias** dessa função para diferentes tipos pertencentes à classe.
- Os valores também podem ter restrição de classe:

```
1 3 :: Num a => a
```

O que resolve nosso problema anterior.

Lembrando:

- **Tipo:** coleção de valores relacionados.
- **Classe:** coleção de tipos que suportam certas funções ou operadores.
- **Métodos:** funções requisitos de uma classe.

- Tipos que podem ser comparados por igualdade e desigualdade:

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

```
1 > 1 == 2
2 False
3 > [1,2,3] == [1,2,3]
4 True
5 > "Ola" /= "Alo"
6 True
```

- A classe Eq acrescida de operadores² de ordem:

```
1 (<)  :: a -> a -> Bool
2 (<=) :: a -> a -> Bool
3 (>)  :: a -> a -> Bool
4 (>=) :: a -> a -> Bool
5 min  :: a -> a -> a
6 max  :: a -> a -> a
```

²Quando a função é comumente utilizada de maneira infix, ela também é usualmente chamada de operador.

```
1 > 4 < 6
2 > min 5 0
3 > max 'c' 'h'
4 > "Ola" <= "Olaf"
```

- A classe `Show` define como converter um valor de um tipo para uma representação em uma string:

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

```
1 > show 10.0
2 > show [1,2,3,4]
```

- A classe `Read` define como ler um valor de uma `String`:

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

- Precisamos especificar o tipo que queremos extrair da String:

```
1 > read "12.5" :: Double
2 > read "False" :: Bool
3 > read "[1,3,4]" :: [Int]
```

- Em geral a função **read** é a inversa da função **show**. Contudo, se **read (show x) == x** é verdadeiro depende da implementação e do tipo em questão.

- A classe `Num` define todos os tipos numéricos e deve suas instâncias devem implementar:

```
1 (+) :: a -> a -> a
2 (-) :: a -> a -> a
3 (*) :: a -> a -> a
4 negate :: a -> a
5 abs :: a -> a
6 signum :: a -> a
7 fromInteger :: Integer -> a
```

```
1 > 1 + 3
2 > 6 - 9
3 > 12.3 * 5.6
```

- O que as seguintes funções fazem? (use o `:t` para ajudar)

```
1 > negate 2
2 > abs 6
3 > signum (-1)
4 > fromInteger 3
```

- **negate**: inverte o sinal do argumento.
- **abs**: retorna o valor absoluto.
- **signum**: retorna o sinal do argumento.
- **fromInteger**: converte um argumento do tipo inteiro para numérico.

Note que os valores negativos devem ser escritos entre parênteses para não confundir com o operador de subtração.

- A classe `Integral` define todos os tipos numéricos inteiros e suas instâncias devem implementar:

```
1 quot :: a -> a -> a
2 rem  :: a -> a -> a
3 div  :: a -> a -> a
4 mod  :: a -> a -> a
5 quotRem :: a -> a -> (a, a)
6 divMod  :: a -> a -> (a, a)
7 toInteger :: a -> Integer
```

- Envolver uma função entre acentos graves transforma uma função em operador infix.

```
1 > quot 10 3 == 10 `quot` 3
```

```
1 > 10 `quot` 3
2 > 10 `rem` 3
3 > 10 `div` 3
4 > 10 `mod` 3
```

Pergunta

Pra que 2 de cada?

- As funções `quot` e `rem` arredondam para o 0, enquanto `div` e `mod` para $-\infty$

- A classe `Fractional` define todos os tipos numéricos fracionários e suas instâncias devem responder:

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

```
1 > 10 / 3
2 > recip 10
```

Qual a diferença entre esses dois operadores de exponenciação?

-
- 1 `(^)` `:: (Num a, Integral b) => a -> b -> a`
 - 2 `(**)` `:: Floating a => a -> a -> a`
-

```
1 class Fractional a => Floating a where
2   pi :: a
3   exp :: a -> a
4   log :: a -> a
5   sqrt :: a -> a
6   (**) :: a -> a -> a
7   logBase :: a -> a -> a
```

```
1 sin :: a -> a
2 cos :: a -> a
3 tan :: a -> a
```

```
1 asin :: a -> a
2 acos :: a -> a
3 atan :: a -> a
```

```
1 sinh :: a -> a
2 cosh :: a -> a
3 tanh :: a -> a
4 asinh :: a -> a
5 acosh :: a -> a
6 atanh :: a -> a
```

- No `ghci`, o comando `:info` mostra informações sobre os tipos e as classes de tipo:

```
1 > :info Integral
2 class (Real a, Enum a) => Integral a where
3   quot :: a -> a -> a
4   rem  :: a -> a -> a
5   div  :: a -> a -> a
6   mod  :: a -> a -> a
7   quotRem :: a -> a -> (a, a)
8   divMod  :: a -> a -> (a, a)
9   toInteger :: a -> Integer
10  {-# MINIMAL quotRem, toInteger #-}
```

- No `ghci`, o comando `:info` mostra informações sobre os tipos e as classes de tipo:

```
1 > :info Bool
2 data Bool = False | True    -- Defined in 'GHC.Types'
3 instance Eq Bool -- Defined in 'GHC.Classes'
4 instance Ord Bool -- Defined in 'GHC.Classes'
5 instance Show Bool -- Defined in 'GHC.Show'
6 instance Read Bool -- Defined in 'GHC.Read'
7 instance Enum Bool -- Defined in 'GHC.Enum'
8 instance Bounded Bool -- Defined in 'GHC.Enum'
```

- Escreva as definições para os seguintes tipos em um arquivo `atividade02.hs` e carregue no `ghci`. Não importa o que ela faça, só não pode gerar erro:

```
1 bools :: [Bool]
2 nums  :: [[Int]]
3 soma  :: Int -> Int -> Int -> Int
4 copia :: a -> (a, a)
5 f     :: a -> a
6 g     :: Eq a => a -> (a, a) -> Bool
7 h     :: Num a => Int -> a -> a
```

```
1 bools :: [Bool]
2 bools = [True, False]
3
4 nums  :: [[Int]]
5 nums = [[1,2,3],[4,5,6]]
6
7 soma  :: Int -> Int -> Int -> Int
8 soma x y z = x+y+z
```

```
1 copia :: a -> (a, a)
2 copia x = (x,x)
3
4 f :: a -> a
5 f x = x
6
7 g :: Eq a => a -> (a,a) -> Bool
8 g x (y,z) = (x==y) || (x==z)
9
10 h :: Num a => Int -> a -> a
11 h x y = fromIntegral x * y
```
