Paradigmas de Programação

Fabrício Olivetti de França

12 e Junho de 2018

Tipos e Classes padrões

Tipos de dados

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

Tipos de dados

No Haskell, os tipos são definidos pela notação

```
v :: T
```

significando que *v* define um valor do tipo *T*.

Tipos de dados

```
False :: Bool
True :: Bool
10 :: Int
```

Tipos de funções

De forma similar uma função pode ser definida por

```
f :: T0 -> T1
```

indicando que a função *f* recebe um valor do tipo *T0* e retorna um valor do tipo *T1*.

Tipos avaliados

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

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

Inferência de tipo

No Haskell, toda expressão tem um tipo calculado antes de avaliar o resultado da expressão.

Os tipos podem ser definidos automaticamente pela inferência do tipo.

Inferência de tipo

Se eu tenho:

```
f :: A -> B
e :: A
```

então

```
f e :: B
```

Exemplo

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

Qual o tipo da função?

Exemplo

Abra o **ghci** e digite:

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

Exemplo

Abra o **ghci** e digite:

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

Exemplo

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

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

Exemplo

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

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

Exemplo

A assinatura da função fica:

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

Exemplo

Se eu fizer (ou tentar):

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

Isso vai gerar um erro de compilação!!

Exemplo

• 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

Tipos Básicos

O compilador GHC já vem com suporte nativo a diversos tipos básicos para uso.

Durante o curso veremos como definir alguns deles.

Tipos Básicos

Os tipos são:

- **Bool:** contém os valores **True** e **False**. Expressões booleanas podem ser executadas com os operadores `&&` (e), `||` (ou) e `not`.
- **Char:** contém todos os caracteres no sistema **Unicode**. Podemos representar a letra ‘a’, o número ‘5’ e a seta tripla ‘⇶’.
- **String:** sequências de caracteres delimitados por aspas duplas: “Olá Mundo”.

Tipos Básicos

- **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.

Tipos Básicos

Note que ao escrever:

```
x = 3
```

O tipo de *x* pode ser *Int, Integer, Float* e *Double*. Qual tipo devemos atribuir a *x*?

Listas

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

```
[1,2,3,4]
```

Listas

Uma lista de tipo T tem tipo [T]:

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

Listas

O tamanho da lista (*length*) representa a quantidade de elementos nela. Um lista vazia é representada por `[]` e listas com um elemento, como `[1]`, `[False]`, `[[]]` são chamadas *singleton*.

Listas

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

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

Listas

Notem que:

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

Tuplas

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

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

O tipo da tupla é definido como `(T1, T2,...,Tn)`.

Tuplas

O número de componentes de uma lista é chamado *aridade*. Uma tupla de aridade zero, a tupla vazia, é representada por `()`, tuplas de tamanho dois são conhecidas como *duplas*, tamanho três são *triplas*.

Tuplas

Notem que:

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

Funções

Funções são mapas de argumentos de um tipo para resultados em outro tipo. O tipo de uma função é escrita como `T1 -> T2`, ou seja, o mapa do tipo `T1` para o tipo `T2`:

```
not  :: Bool -> Bool
even :: Int -> Bool
```

Funções

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. Criem as seguintes funções em um arquivo *aula02.hs*, carreguem no *ghci* e verifiquem seu tipo e testem com algumas entradas:

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

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

Funções

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:

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

Curry

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

Curry

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

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

Curry

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

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

Curry

A seguinte definição ainda é válida:

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

Curry

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**.

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

Curry

Da mesma forma podemos ter:

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

Curry

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

```
soma' :: Int -> Int -> Int
soma' x y = x + y

mult :: Int -> Int -> Int -> Int
mult x y z = x*y*z
```

Polimorfismo

Tipos polimórficos

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

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

Tipos polimórficos

Qual o tipo de `length`?

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

Tipos polimórficos

Qual o tipo de `length`?

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

Quem é `a`?

Tipos polimórficos

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`, ...

Overloaded types

Considere agora a função `(+)`, diferente de `length` ela pode ter um comportamento diferente para tipos diferentes.

Internamente somar dois `Int` pode ser diferente de somar dois `Integer`. De todo modo, essa função **deve** ser aplicada a tipos numéricos.

Overloaded types

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

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

O operador + recebe dois tipos de uma classe numérica e retorna um valor desse tipo.

Overloaded types

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

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

Overloaded types

Uma vez que uma função contém uma restrição de classe, pode ser necessário definir **instâncias** dessa função para diferentes tipos pertencentes a classe.

Os valores também podem ter restrição de classe:

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

resolvendo nosso problema anterior.

Classes de tipos

Lembrando:

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

Eq - classe da igualdade

Tipos que podem ser comparados em igualdade e desigualdade:

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

Eq - classe da igualdade

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

Ord - classe de ordem

A classe Eq acrescido de operadores de ordem:

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

Ord - classe de ordem

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

Show - classe imprimíveis

A classe `Show` define como imprimir um valor de um tipo:

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

Show - classe imprimíveis

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

Read - classe legíveis

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

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

Read - classe legíveis

Precisamos especificar o tipo que queremos extrair da String:

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

Num - classe numérica

A classe `Num` define todos os tipos numéricos e deve ter as instâncias de:

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

Num - classe numérica

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

Num - classe numérica

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

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

Num - classe numérica

- **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.

Integral - classe de números inteiros

A classe `Integral` define todos os tipos numéricos inteiros e deve ter as instâncias de:

```
quot :: a -> a -> a
rem :: a -> a -> a
div :: a -> a -> a
mod :: a -> a -> a
quotRem :: a -> a -> (a, a)
divMod :: a -> a -> (a, a)
toInteger :: a -> Integer
```

Integral - classe de números inteiros

O uso de crases transforma uma função em operador infixo.

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

Integral - classe de números inteiros

```
> 10 `quot` 3
> 10 `rem` 3
> 10 `div` 3
> 10 `mod` 3
```

Integral - classe de números inteiros

As funções `quot` e `rem` arredondam para o 0, enquanto `div` e `mod` para o infinito negativo.

Fractional - classe de números inteiros

A classe `Fractional` define todos os tipos numéricos fracionários e deve ter as instâncias de:

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

Fractional - classe de números inteiros

```
> 10 / 3
> recip 10
```

Outros operadores e funções úteis

Qual a diferença entre esses dois operadores de exponenciação?

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

Floating - classe de números de ponto flutuante

```haskell
class Fractional a => Floating a where
  pi :: a
  exp :: a -> a
  log :: a -> a
  sqrt :: a -> a
  (**) :: a -> a -> a
  logBase :: a -> a -> a
```

Floating - classe de números de ponto flutuante

```
sin :: a -> a
cos :: a -> a
tan :: a -> a
```

Floating - classe de números de ponto flutuante

```
asin :: a -> a
acos :: a -> a
atan :: a -> a
```

Floating - classe de números de ponto flutuante

```
sinh :: a -> a
cosh :: a -> a
tanh :: a -> a
asinh :: a -> a
acosh :: a -> a
atanh :: a -> a
```

Info

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

```
> :info Integral
class (Real a, Enum a) => Integral a where
  quot :: a -> a -> a
  rem :: a -> a -> a
  div :: a -> a -> a
  mod :: a -> a -> a
  quotRem :: a -> a -> (a, a)
  divMod :: a -> a -> (a, a)
  toInteger :: a -> Integer
  {-# MINIMAL quotRem, toInteger #-}
```

Info

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

```
> :info Bool
data Bool = False | True     -- Defined in ‘GHC.Types’
instance Eq Bool -- Defined in ‘GHC.Classes’
instance Ord Bool -- Defined in ‘GHC.Classes’
instance Show Bool -- Defined in ‘GHC.Show’
instance Read Bool -- Defined in ‘GHC.Read’
instance Enum Bool -- Defined in ‘GHC.Enum’
instance Bounded Bool -- Defined in ‘GHC.Enum’
```

Atividade

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:

```
bools :: [Bool]
nums :: [[Int]]
soma :: Int -> Int -> Int -> Int
copia :: a -> (a, a)
f :: a -> a
g :: Eq a => a -> (a, a) -> Bool
h :: Num a => Int -> a -> a
```