

Linguagens de Programação

Programação Funcional e Haskell
Tipos e Currificação
Thiago Alves

O que é um Tipo?

Um tipo é um nome para uma coleção de valores relacionados. Por exemplo, em Haskell o tipo básico

Bool

Contém dois valores lógicos:

False

True

Erros de Tipo

Aplicar a função a um ou mais argumentos do tipo errado é chamado de erro de tipo.

```
> 1 + False  
ERROR
```

1 é um número e False é um valor lógico, mas + requer dois números.

Tipos em Haskell

- Se a avaliação de uma expressão e produzir um valor de tipo t, então e tem tipo t, escrito:

`e :: t`

- Toda expressão bem formada tem um tipo. O tipo é calculado tempo de compilação usando um processo chamado inferência de tipo.

Inferência de Tipos

- Regras para aplicações de funções

$$\frac{f :: a \rightarrow b \quad e :: a}{f e :: b}$$

- Exemplo:

$$\frac{\text{not} :: \text{Bool} \rightarrow \text{Bool} \quad \text{False} :: \text{Bool}}{\text{not False} :: \text{Bool}}$$

Inferência de Tipos

- Programas em Haskell são type safe
- Em geral, erros de tipo não ocorrem durante a execução
- Todos os erros de tipo são encontrados em tempo de compilação

Inferência de Tipos

- Isso torna os programas mais seguros e mais rápidos
- Tipo correto em tempo de execução
- Sem checagem de tipos em tempo de execução

- No GHCi, o comando `:type` calcula o tipo de uma expressão, sem realizar a expressão:

```
> not False  
True
```

```
> :type not False  
not False :: Bool
```


Tipos Básicos

Haskell tem tipos básicos, incluindo:

Bool

- valores lógicos

Char

- caracteres únicos

String

- strings de caracteres

Int

- inteiros de precisão fixa

Integer

- inteiros de precisão arbitrária

Float

- números ponto-flutuante

Tipo Lista

Um list é uma sequência de valores do mesmo tipo:

```
[False,True,False] :: [Bool]
```

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

Em geral:

[t] é o tipo de listas com elementos de tipo t.

Nota:

- O tipo de uma lista não diz nada sobre seu tamanho:

```
[False,True]    :: [Bool]
```

```
[False,True,False] :: [Bool]
```

- O tipo dos elementos não tem restrições. Por exemplo, podemos ter listas de listas:

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

Tipo Tupla

Uma tupla é uma sequência de valores de tipos que podem ser diferentes:

```
(False,True)    :: (Bool,Bool)
```

```
(False,'a',True) :: (Bool,Char,Bool)
```

Em geral:

(t_1, t_2, \dots, t_n) é o tipo das n -tuplas cujo i -ésimo componente tem tipo t_i para i em $1 \dots n$.

Nota:

- O tipo de uma tupla codifica seu tamanho:

```
(False,True)      :: (Bool,Bool)
```

```
(False,True,False) :: (Bool,Bool,Bool)
```

- O tipo dos componentes não tem restrições:

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

```
(True,['a','b'])  :: (Bool,[Char])
```

Tipo Função

Uma função é um mapeamento de valores de um tipo para valores de outro tipo:

```
not :: Bool → Bool
```

```
even :: Int → Bool
```

Em geral:

$t1 \rightarrow t2$ é o tipo de funções que mapeiam valores de tipo $t1$ para valores de tipo $t2$.

Nota:

- Os tipos dos argumentos e dos resultados não têm restrições. Por exemplo, funções com múltiplos argumentos ou resultados são possíveis usando listas ou tuplas:

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

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

Funções Currificadas

Funções com múltiplos argumentos também são possíveis retornando funções como resultados:

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

add' recebe um inteiro x e retorna uma função add' x. Essa função recebe um inteiro y e retorna o resultado x+y.

Nota:

- `add` e `add'` produzem o mesmo resultado final, mas `add` recebe seus dois argumentos ao mesmo tempo, enquanto `add'` recebe eles um por vez:

```
add :: (Int,Int) → Int
```

```
add' :: Int → (Int → Int)
```

- Funções que recebem seus argumentos um por vez são chamadas de funções currificadas, celebrando o trabalho de Haskell Curry em tais funções.

- Funções com mais que dois argumentos podem ser currificadas retornando funções aninhadas:

```
mult    :: Int → (Int → (Int → Int))  
mult x y z = x*y*z
```

mult recebe um inteiro x e retorna uma função mult x, que por sua vez recebe um inteiro y e retorna uma função mult x y, que finalmente recebe um inteiro z e retorna o resultado $x*y*z$.

Utilidade da Currificação?

Funções currificadas são mais flexíveis que funções em tuplas porque funções úteis podem ser definidas pela aplicação parcial de uma função currificada.

Por exemplo:

```
add' 1 :: Int => Int
```

```
take 5 :: [Int] -> [Int]
```

```
drop 5 :: [Int] -> [Int]
```

Convenções de Currificação

Para evitar excesso de parênteses no uso de funções currificadas, duas simples convenções são usadas:

- A flecha \rightarrow é associativa pela direita.

$\text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$

Significa $\text{Int} \rightarrow (\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}))$.

- Como consequência, é natural para aplicações de funções serem associadas pela esquerda.

mult x y z



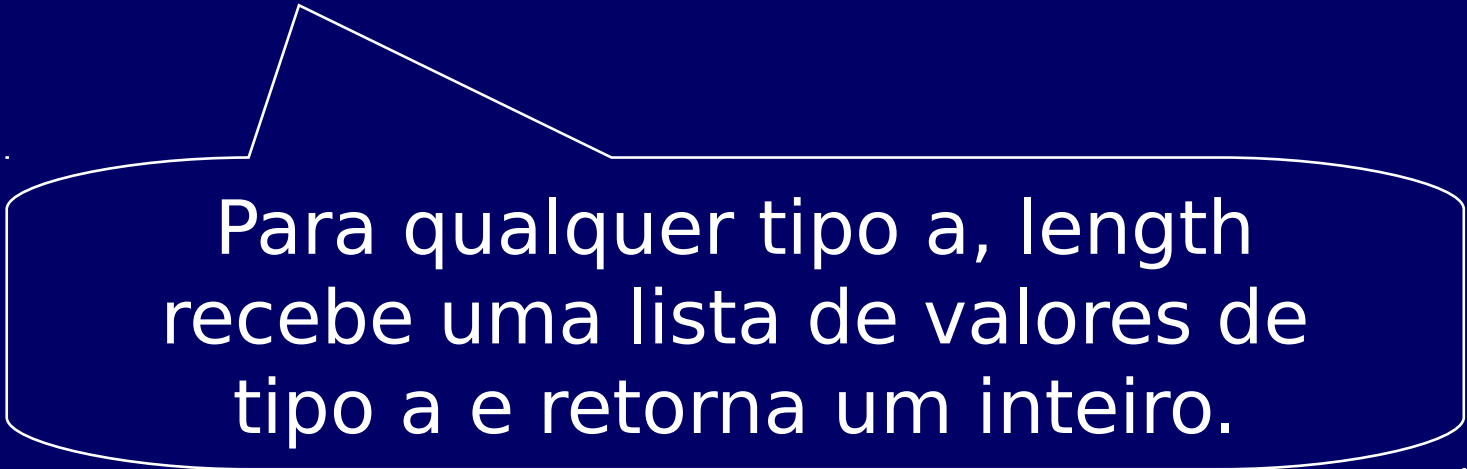
Significa ((mult x) y) z.

A menos que as tuplas sejam explicitamente requeridas, todas as funções em Haskell são definidas na forma currificada.

Funções Polimórficas

Uma função é chamada polimórfica se seu tipo contém uma ou mais variáveis de tipo.

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



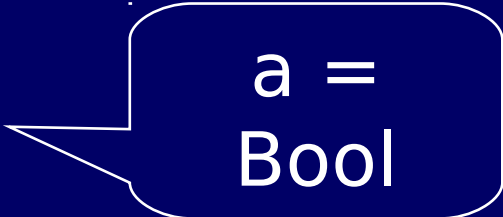
Para qualquer tipo a , `length` recebe uma lista de valores de tipo a e retorna um inteiro.

Nota:

- Variáveis de tipo podem ser instanciadas com tipos diferentes em circunstâncias diferentes:

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

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



a =
Bool



a = Int

- Variáveis de tipo devem começar com letra minúscula e normalmente chamadas a, b, c, etc.

- Várias funções definidas na biblioteca padrão são polimórficas. Por exemplo:

```
fst :: (a,b) -> a
```

```
head :: [a] -> a
```

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

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

```
id   :: a -> a
```


Exercícios

(1) Qual o tipo do seguinte valor?

```
[tail,init,reverse]
```

(1) Qual o tipo do seguinte valor?

`[tail,init,reverse] :: [[a] → [a]]`

(2) Qual o tipo da seguinte função?

```
twice f x = f (f x)
```

(2) Quais os tipos das seguintes funções?

```
twice f x = f (f x) :: (a → a) → a → a
```

Sobrecarga de Tipos

O operador aritmético $+$ calcula a soma de dois valores de tipos números

```
> 1 + 2  
3
```

```
> 1.1 + 2.2  
3.3
```

Sobrecarga de Tipos

Restrições de classe no tipo para deixar claro.

Um tipo é sobrecarregado se seu tipo contém uma ou mais restrições de classe.

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

Para qualquer tipo numérico a , $(+)$ recebe dois valores de tipo a e retorna um valor de tipo a .

Nota:

- Variáveis de tipo restringidas podem ser instanciadas para quaisquer tipos que satisfazem as restrições:

```
> 1 + 2  
3
```

```
> 1.0 + 2.0  
3.0
```

```
> 'a' + 'b'  
ERROR
```

a = Int

a =
Float

Char is not
a numeric
type

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

- Variáveis de tipo restringidas podem ser instanciadas para quaisquer tipos que satisfazem as restrições:

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

$a = \text{Int}$

```
> sum [1.1,1.2]  
2.3
```

$a = \text{Float}$

```
> sum ['a']  
ERROR
```

Char is not
a numeric
type

- Classes são conjuntos de tipos que suportam certas operações sobrecarregadas
- Haskell tem várias classes básicas:

Num - Tipos numéricos

Eq - Tipos de igualdade

Ord - Tipos ordenados

- Classe Eq possui os tipos em que seus valores podem ser comparados

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

```
> False == False  
True
```

```
> [1,2] == [1,2,3]  
False
```

- Em geral, funções não estão na classe Eq.

- Classe Ord possui os tipos em que seus valores são totalmente ordenados

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

```
> False < True  
True
```

```
> [1,2,3] < [1,2]  
False
```

```
> 'a' < 'b'  
True
```

- Classe Num possui os tipos em que seus valores são numéricos

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

```
negate :: Num a => a -> a
```

```
signum :: Num a => a -> a
```

```
> 1 + 2.3  
3.3
```

```
> negate 1.3  
-1.3
```

```
> signum -3  
-1
```