Tipos em Haskell

Programação Funcional

Prof. Rodrigo Ribeiro

Objetivos

- Conceitos sobre tipos em linguagens de programação.
- Definições de polimorfismo paramétrico e de sobrecarga.
- Verificação e inferência de tipos em Haskell.

Setup Inicial

```haskell
module Aula03 where

main :: IO ()
main = return ()
```

Haskell

- Expressões são o componente básico de programas funcionais.
- Em Haskell, toda expressão possui um tipo.
- Mas o que é um tipo?

Expressões e Tipos

- Tipo: Coleção de valores que suportam o mesmo conjunto de operações.
- Definimos que uma expressão possui um certo tipo como `expression :: Type`.

Expressões e Tipos

- Exemplos:

```
True  :: Bool
'a'   :: Char
[1,2] :: [Int]
(1, True) :: (Int, Bool)
not :: Bool -> Bool
```

Expressões e Tipos

- Aplicações de função também possuem tipos.

```
1 + 2     :: Int
not True  :: Bool
```

Tipagem Estática

- Haskell não permite a execução de código com erros de tipo.
  - Isto é, Haskell possui tipagem estática.
  - Algumas linguagens são *dinamicamente* tipadas: Ruby, Python, JavaScript, etc. . .

Tipagem Estática

- Consequência: nenhuma falha em tempo de execução é decorrente de erro de tipos.
  - Programas Haskell são *type safe*.
- Alguns programas válidos são rejeitados pois não é possível avaliar o código para definir se este é seguro.

```
if True then 1 else  False
```

Verificação de tipos

- Regra geral: se `f :: A -> B` e `x :: A` então `f x :: B`.
- Usando essa regra, podemos deduzir que uma expressão não é válida. Exemplo:

```
not :: Bool -> Bool
'a' :: Char

not 'a'
-- Couldn't match expected type
-- ‘Bool’ with actual type ‘Char’
```

Importante: Parse errors

- O seguinte tipo de erro é recorrente aprendendo Haskell:

```
isZero x = x = 0
<interactive>:1:14: error:
    parse error on input ‘=’
```

- **Parse error**: código não segue a sintaxe / identação da linguagem.
- Outro erro comum: Tipos, módulos e classes começam com letras maiúsculas. Variáveis e funções, letras minúsculas.

Tipos básicos

- `Bool`: Valores lógicos - `True, False`.
- `Char`: caracteres simples - `'a'`.
- Tipos integrais.
  - `Int`: inteiros com precisão limitada (pela máquina).
  - `Integer`: inteiros com precisão ilimitada.
- Tipos de ponto flutuante.
  - `Float`: precisão simples.
  - `Double`: precisão dupla.

Tipos compostos

- Listas [T]: sequências homogênas de valores de um tipo `T`.
- Tuplas de diferentes aridades.
  - pares `(T1, T2)`
  - triplas `(T1, T2, T3)`
  - `...` até 62 componentes `(T1, ... , T62)`.
- Funções: T1 -> T2 -> T3 ... -> R

Algumas diferenças

- Lista de tuplas e tuplas de listas:

```
([1,2],[True]) :: ([Int], [Bool])
[(1, True),(2, False)] :: [(Int,Bool)]
```

- Funções e pares

```
f :: Int -> Int -> Int
-- f recebe dois argumentos
g :: (Int, Int) -> Int
-- g recebe um argumento,
-- que é um par
f 1 2     -- ok
g (1, 2)  -- ok
g 1 2     -- error...
```

Funções são cidadãos de 1a classe

```haskell
-- funções como elementos de uma lista
[(+), (*), (-)] :: [Int -> Int -> Int]
[(&&), (||)]    :: [Bool -> Bool -> Bool]

-- Elementos devem possuir o mesmo tipo.
-- [(+), (&&)] -- erro de tipo!

-- funções podem ser passadas e retornadas
-- como resultados de outras funções

flip :: (a -> b -> c) -> (b -> a -> c)
```

Polimorfismo

- Haskell provê suporte a dois tipos de polimorfismo: paramétrico e sobrecarga.
- Polimorfismo paramétrico permite a definição de código que opera da mesma forma sobre valores de tipos diferentes.
- Polimorfismo de sobrecarga permite a definição de código que opera de maneira distinta de acordo com o tipo de valores.

Polimorfismo paramétrico

- Funções operam sobre "todos" os tipos.
- Tipos envolvem variáveis: identificadores formados por letras minúsculas.
- Exemplo:

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

length [1, 2]      -- Ok, a = Int
length ['a', 'b'] -- Ok, a = Char
length [True]      -- Ok, a = Bool
```

Mais funções polimórficas

```
null :: [a] -> Bool
(++) :: [a] -> [a] -> [a] -- concatenação
reverse :: [a] -> [a]
```

- **Importante!** Variáveis de tipo devem ser substituídas de maneira uniforme. Exemplo:

```
[1, 2] ++ [3, 4] -- Ok, a é substituído por Int
[1, 2] ++ ['a', 'b'] -- Erro!
```

Inferência de tipos

- Processo no qual o compilador é capaz de deduzir o tipo de uma definição.
- Em Haskell, o GHC é capaz de calcular o tipo "mais polimórfico" para qualquer expressão.

Inferência de tipos

- Exemplo: Determinar o tipo da seguinte função.

```
id x = x
```

Inferência de tipos

- Qual o tipo de `id`?

```
id x = x
```

- É uma função de um argumento.
  - Logo, seu tipo deve ser `?1 -> ?2` para tipos `?1` e `?2`.
- É uma função que retorna o seu argumento como resultado.
  - Logo, temos que `?1 = ?2`.
- Não há nenhuma restrição adicional.
  - Logo, o tipo é `?1 -> ?1`, que é generalizado para `a -> a`.

Exemplo

- Qual o tipo de `id id`?

Exemplo

- Qual o tipo de `id id`?
  - Lembre-se: `id :: a -> a`
- Atribuindo variáveis diferentes para ocorrências diferentes.
  - Primeiro `id : ?1 -> ?1`
  - Segundo `id: ?2 -> ?2`

Exemplo

- Qual o tipo de `id id`?
  - Lembre-se: `id :: a -> a`
- Atribuindo variáveis diferentes para ocorrências diferentes.
  - Primeiro `id : ?1 -> ?1`
  - Segundo `id: ?2 -> ?2`
- Se `f :: A -> B` em `f x` então `x :: A`
  - No exemplo, `f x = id id`, então `f = x = id`.
  - Logo, temos que `?1 = ?2 -> ?2`.
  - Logo, o primeiro `id` possui o tipo `(?2 -> ?2) -> (?2 -> ?2)`.

Exemplo

- Qual o tipo de `id id`?
  - Lembre-se: `id :: a -> a`
- Atribuindo variáveis diferentes para ocorrências diferentes.
  - Primeiro `id : ?1 -> ?1`
  - Segundo `id: ?2 -> ?2`
- Se `f :: A -> B` em `f x` então `x :: A`
  - No exemplo, `f x = id id`, então `f = x = id`.
  - Logo, temos que `?1 = ?2 -> ?2`.
  - Logo, o primeiro `id` possui o tipo `(?2 -> ?2) -> (?2 -> ?2)`.
- O tipo do resultado `f x` é `f x :: B`.
  - Neste caso, `B = ?2 -> ?2`.
  - Portanto, `id id : ?2 -> ?2`.

Exemplo

- Qual o tipo de `id id`?
  - Lembre-se: `id :: a -> a`
- Atribuindo variáveis diferentes para ocorrências diferentes.
  - Primeiro `id : ?1 -> ?1`
  - Segundo `id: ?2 -> ?2`
- Se `f :: A -> B` em `f x` então `x :: A`
  - No exemplo, `f x = id id`, então `f = x = id`.
  - Logo, temos que `?1 = ?2 -> ?2`.
  - Logo, o primeiro `id` possui o tipo `(?2 -> ?2) -> (?2 -> ?2)`.
- O tipo do resultado `f x` é `f x :: B`.
  - Neste caso, `B = ?2 -> ?2`.
  - Portanto, `id id : ?2 -> ?2`.
- Como não há restrições adicionais, o tipo final é generalizado.
  - `id id : a -> a`.

Listas

- Elementos de uma lista devem ser de um mesmo tipo.
- Exemplo:

```
sin :: Float -> Float
[sin , id] :: [Float -> Float]
```

Exemplo

- Considere os seguintes tipos.

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

A seguintes expressão é válida?

```
[head, length]
```

Exemplo

- Para ser válida, ambos os elemento devem ter o mesmo tipo.
- Logo, temos que os seguintes tipos devem ser iguais:

```
[?1] -> ?1 = [?2] -> Int
```

- Note que para a igualdade anterior ser verdadeira, temos que `?1 = ?2` e `?1 = Int`.
- Substituindo, chegamos no tipo `[Int] -> Int` para ambos os elementos.

Sobrecarga

- Em Haskell, a adição opera sobre diferentes tipos.

```
1 + 2 -- integers
```

```
2.5 + 3.1 -- floating point
```

- Mas, a adição não é definida sobre todo tipo.

```
'a' + 'b'
No instance for (Num Char)
arising from a use of '+'
```

Sobrecarga

- Não é possível atribuir o seguinte tipo à adição:

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

porquê a adição não é definida para todo tipo.

Sobrecarga

- Vamos usar o interpretador de Haskell para descobrir o tipo da adição.

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

- O termo `Num a` antes do símbolo => é uma **restrição**.
- Restringe $(+)$ a tipos que satisfazem essa restrição.
  - Neste caso, a restrição é que a deve ser um tipo "numérico".
- **Num** é uma **classe de tipos**
  - **Aviso!** Conceito não relacionado a OO.

Classes de tipos

- De maneira simples, classes de tipo definem um conjunto de operações suportador por certos tipos ditos instâncias desta classe.
- Diversas operações da biblioteca padrão de Haskell utilizam classes de tipos.
- O tópico de classes de tipos será estudado com detalhes quando abordarmos o conceito de sobrecarga.

Algumas classes básicas

- A classe `Num` define uma interface para tipos numéricos.
  - Operações incluem `(+)`, `(*)` e `abs`.
  - Tipos que são instâncias desta classe são `Int`, `Double`, `Float`, `Integer`.
  - Os tipos `Bool`, `Char` e listas não são instâncias de `Num`.

Algumas classes básicas

- A classe `Eq` define uma interface para tipos que suportam teste de igualdade.

```
(==) :: Eq a => a -> a -> Bool -- igual
(/=) :: Eq a => a -> a -> Bool -- diferente
```

- Os tipos numéricos, `Bool`, `Char`, listas e tuplas são instâncias de `Eq`.
- Tipos funcionais não são instâncias de `Eq`.

Algumas classes básicas

- A classe `Ord` define uma interface para tipos que suportam operações de comparação.

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

- Os tipos numéricos, `Bool`, `Char`, listas e tuplas são instâncias de `Ord`.
- Tipos funcionais não são instâncias de `Ord`.

Algumas classes básicas

- A classe `Show` define uma operação que converte valores em `Strings`.

```
show :: Show a => a -> String
```

- Quase todos os tipos podem ser instâncias de `Show`.
- Tipos funcionais não são instâncias de `Show`.

Finalizando

- Toda expressão possui um tipo.
- Tipos são usados de duas maneiras:
  - Verificação de tipos
  - Inferência de tipos
- Haskell possui duas formas de polimorfismo:
  - Polimorfismo paramétrico.
  - Polimorfismo de sobrecarga.

Exercícios

- Escreva definições que possuam os seguintes tipos. Não se preocupe se seu código faz ou não sentido desde que ele seja aceito pelo compilador.

```
bools :: [Bool]
nums  :: [[Int]]
add   :: Int -> Int -> Int -> Int
copy  :: a -> (a, a)
apply :: (a -> b) -> a -> b
swap  :: (a,b) -> (b,a)
```