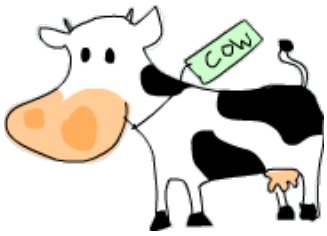
Começando Índice Sintaxe em Funções

Tipos e Typeclasses

Acredite no tipo



Já falamos que Haskell possui um sistema de tipos estático. O tipo de toda expressão é conhecido na hora da compilação, o que resulta num código mais seguro. Se você escrever um programa que tente dividir um tipo booleano por um número, ele nem compilará. Isso é bom porque é melhor detectarmos erros logo ao terminar de programar do que se deparar com travamentos indesejados. Tudo em Haskell tem um tipo, então o compilador pode considerar várias possibilidades antes mesmo de compilá-lo.

Ao contrário de Java ou Pascal, Haskell tem inferência de tipo. Se for digitado um número, não precisamos avisar ao Haskell que é um número. Ele consegue identificar isso automaticamente, não damos os tipos de funções e expressões explicitamente. Nós veremos apenas o básico de Haskell com uma visão apenas superficial sobre tipos. No entanto, entender o sistema de tipos é muito importante para o aprendizado de Haskell.

Um "tipo" é algo como uma etiqueta que toda expressão têm, que nos diz em qual categoria ela se encaixa. A expressão **`True`** é booleana, **`"hello"`** é uma string, etc.

Agora usaremos o GHCI para descobrir os tipos de algumas expressões. Faremos isso usando o comando **`:t`** que, seguido de qualquer expressão válida, retorna o seu tipo. Vamos dar uma olhada.

```
ghci> :t 'a'
'a' :: Char
ghci> :t True
True :: Bool
ghci> :t "HELLO!"
"HELLO!" :: [Char]
ghci> :t (True, 'a')
(True, 'a') :: (Bool, Char)
ghci> :t 4 == 5
4 == 5 :: Bool
```

Fazendo isso, vemos que **`:t`** mostra a expressão seguida de **`::`** e seu tipo. **`::`** pode ser lido como "do tipo". Tipos explícitos sempre são referidos pela sua primeira letra maiúscula. **`'a'`**, como é visto, é do tipo **`Char`**. Não é difícil de identificar que vem da palavra *caracter*. **`True`** é **`Boolean`**. Faz sentido. Mas, o que é isso? Examinando o tipo de **`"HELLO!"`** descobrimos que ele é **`[Char]`**. Os colchetes denotam uma lista. Então lemos isso como uma *lista de caracteres*. Ao contrário de listas, cada valor da tupla tem seu tipo. Então a expressão **`(True, 'a')`** tem o tipo **`(Bool, Char)`**, e uma expressão como **`('a','b','c')`** deverá retornar **`(Char, Char, Char)`**. **`4 == 5`** sempre retornará **`False`**, que é do tipo **`Bool`**.

Funções também têm tipos. Quando escrevemos nossas próprias funções, podemos declarar explicitamente quais são os seus tipos. Isso geralmente é considerado uma boa prática exceto quando a função é muito curta. Daqui pra frente

daremos vários exemplos que fazem uso de declaração de tipos. Você ainda se lembra daquela compreensão de lista que criamos no capítulo anterior e que usava filtros para retornar somente a parte maiúscula de uma string? Então, aqui esta ela novamente com os tipos declarados.

```
removeNonUppercase :: [Char] -> [Char]
removeNonUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]
```

`removeNonUppercase` tem o tipo **`[Char] -> [Char]`**, o que significa que ele parte de uma string e chega em outra string. Ou seja, ele irá receber como parâmetro uma string e nos devolver outra string como resultado. O tipo **`[Char]`** é sinônimo de **`String`** então ficaria mais claro se fosse escrito **`removeNonUppercase :: String -> String`**. Não precisariamos fazer essa declaração de tipo para o compilador porque ele poderia inferir por si só que essa função recebe uma string e retorna outra string. E se nós precisarmos de uma função que recebe vários tipos como parâmetro? Vamos ver então uma função bem simples que pega três inteiros e os soma:

```
addThree :: Int -> Int -> Int -> Int
addThree x y z = x + y + z
```

Os parâmetros são separados por **`->`** e não há nenhuma distinção entre tipos de parâmetros e retorno. O tipo do retorno é o último e os parâmetros são os três primeiros. Mais adiante veremos o porquê deles serem apenas separados por um **`->`** ao invés de ter um destaque maior como **`Int, Int, Int -> Int`** ou algo do gênero.

Se você deseja especificar o tipo da função, mas não tem certeza de qual deve ser, você pode escrevê-la normalmente e depois descobrir com o **`:t`**. Funções também são expressões, então **`:t`** funciona sem problemas.

Aqui temos um apanhado geral dos principais tipos.

`Int` é inteiro. É usado por números inteiros. **`7`** pode ser um **`Int`** mas **`7.2`** não. **`Int`** possui limitações de tamanho, o que significa ter um máximo e um mínimo. Geralmente os computadores de 32bit têm um **`Int`** máximo de 2147483647 e um mínimo de -2147483648.

`Integer` significa, hmmm... inteiro também. A principal diferença é que não tem limitações e pode ser usado por números realmente grandes. Digo, extremamente grandes. Contudo, **`Int`** é mais eficiente.

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

```
ghci> factorial 50
30414093201713378043612608166064768844377641568960512000000000000
```

`Float` é um número real em ponto flutuante de precisão simples.

```
circumference :: Float -> Float
circumference r = 2 * pi * r
```

```
ghci> circumference 4.0
```

```
25.132742
```

`Double` é um número real em ponto flutuante com o dobro(!) de precisão.

```
circumference' :: Double -> Double
circumference' r = 2 * pi * r
```

```
ghci> circumference' 4.0
25.132741228718345
```

`Bool` é um tipo booleano. Pode ter apenas dois valores: **`True`** ou **`False`**.

`Char` representa um caractere. É delimitado por aspas simples. Uma lista de caracteres é denominada String.

Tuplas são tipos mas possuem uma variação de acordo com a quantidade e valores que contém, então teoricamente temos tuplas com infinitos tipos, o que é demais para cobrir nesse tutorial. Note que uma tupla vazia `()` é também um tipo e pode assumir apenas um valor: **`()`**

Tipo variável

Qual você acha que é o tipo da função **`head`**? Já que **`head`** recebe uma lista e retorna o seu primeiro elemento, qual deve ser o seu tipo? Vamos descobrir!

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

Hmmm! O que é esse **`a`**? É o tipo? Lembre-se que já vimos que o tipo é escrito com a primeira letra maiúscula, então não pode ser exatamente o tipo. E é exatamente essa diferença que nos diz ser um **tipo variável**. Isso significa que o **`a`** pode ser qualquer tipo. Isso é algo como os genéricos de outras linguagens, mas em Haskell é muito mais poderoso porque nos permite facilmente escrever funções mais genéricas caso o processamento seja o mesmo para diferentes tipos. Funções que possuem tipos variáveis são denominadas **funções polimórficas**. A declaração de tipo em **`head`** diz que ele recebe uma lista de elementos de qualquer tipo e retorna um elemento dela.

Embora tipos variáveis possam ter nomes com mais de um caractere, normalmente nós damos a eles nomes como a, b, c, d...

Se lembra da função **`fst`**? Aquela que retorna o primeiro componente de um par? Então, vamos examinar o seu tipo:

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

Examinando o tipo de **`fst`**, a gente vê que ele recebe uma tupla que contém dois tipos e retorna o primeiro elemento. É exatamente por causa disso que conseguimos usar **`fst`** em pares que contenham quaisquer tipos. Perceba ainda que mesmo **`a`** e **`b`** sendo diferentes tipos variáveis, eles não devem ser necessariamente de tipos diferentes. A declaração apenas nos diz que o tipo do primeiro componente (da tupla) deve ser do mesmo que o do retorno.

Basicão de Typeclasses

Uma Typeclass (classe de tipos) é como uma interface que define um comportamento. Se um tipo é parte de uma typeclass, quer dizer que ela suporta e implementa o comportamento especificado pela classe de tipo. Muita gente vinda da orientação a objetos se confunde e acha estar diante de uma classe de OO. Bom... não. Você pode pensar que são como as interfaces de Java, mas na verdade são muito melhor.

Qual deve ser o tipo da função **==**?

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

Nota: o operador de igualdade (**==**) é uma função. Assim como **+**, *****, **-**, **/** e quase todos os outros operadores. Se uma função é composta apenas de caracteres especiais, ela é por padrão uma função infixa. Se quisermos verificar o seu tipo, passe-a para outra função ou chame-a como função prefixa, colocando-a entre parênteses.

Interessante. Temos algo novo aqui, o símbolo **=>**. Tudo antes do símbolo **=>** é denominado **class constraint (restrição de classe)**. Podemos ler a declaração de tipo anterior assim: a função de igualdade recebe dois argumentos de mesmo tipo e retorna um **`Bool`**. Esse tipo deve ser membro da classe **`Eq`** (que é a class constraint).

A typeclass **`Eq`** provê uma interface para o teste de igualdade. Qualquer tipo que faça sentido ser verificado por igualdade com outro tipo deve estar na typeclass **`Eq`**. Todos os tipos Haskell - exceto os de IO (tipo para lidar com entrada e saída) e funções - fazem parte da typeclass **`Eq`**.

A função **`elem`** tem o tipo **`(Eq a) => a -> [a] -> Bool`** porque usa o operador **==** para procurar um determinado elemento em uma dada lista.

Algumas classes de tipo básicas:

`Eq` é usado por tipos que suportam teste por igualdade. As funções que fazem parte dela implementam **==** e **/=**. Se existe alguma class constraint de **`Eq`** para um tipo variável em uma função, usa o operador **==** ou **/=** em algum lugar de sua definição. Todos os tipos já mencionados (com excessão de funções), são parte de **`Eq`**, então podem ser testados por igualdade.

```
ghci> 5 == 5
True
ghci> 5 /= 5
False
ghci> 'a' == 'a'
True
ghci> "Ho Ho" == "Ho Ho"
True
ghci> 3.432 == 3.432
True
```

`Ord` é para tipos que têm ordem.

```
ghci> :t (>)
(>) :: (Ord a) => a -> a -> Bool
```

Todos os tipos já vistos (exceto funções) são parte de **`Ord`**. **`Ord`** engloba todas as funções de comparação comuns como **`>`**, **`<`**, **`>=`** e **`<=`**. A função **`compare`** requer dois membros de **`Ord`** de mesmo tipo e retorna sua ordenação. `Ordering` é uma typeclass que pode ser **`GT`**, **`LT`** ou **`EQ`**, significando *maior que*, *menor que* e *igual a*, respectivamente.

Para ser membro de **`Ord`**, um tipo deve ser membro do prestigioso e restrito clube do **`Eq`**.

```
ghci> "Abrakadabra" < "Zebra"
True
ghci> "Abrakadabra" `compare` "Zebra"
LT
ghci> 5 >= 2
True
ghci> 5 `compare` 3
GT
```

Membros do `Show` podem ser representados como strings. Todos os tipos cobertos até agora (com exceção das funções) são suportados por **`Show`**. A função que lida com a *typeclass* **`Show`** mais usada é a **`show`**. Ela recebe um valor de um que tipo presente em **`Show`** e nos mostra esse valor como uma string.

```
ghci> show 3
"3"
ghci> show 5.334
"5.334"
ghci> show True
"True"
```

`Read` é tipo uma oposição da *typeclass* **`Show`**. A função **`read`** recebe uma string e retorna um tipo membro de **`Read`**.

```
ghci> read "True" || False
True
ghci> read "8.2" + 3.8
12.0
ghci> read "5" - 2
3
ghci> read "[1,2,3,4]" ++ [3]
[1,2,3,4,3]
```

Até agora tudo simples. Todos os tipos já vistos estão nessas classes de tipo. Mas o que acontece ao tentarmos **`read "4"`**?

```
ghci> read "4"
<interactive>:1:0:
    Ambiguous type variable `a' in the constraint:
      `Read a' arising from a use of `read' at <interactive>:1:0-7
    Probable fix: add a type signature that fixes these type variable(s)
```

O que o GHCI está tentando nos dizer é que não sabe o que se esperar como retorno. Perceba que nos usos anteriores de **`read`** nós sempre fazíamos algo com o resultado. Assim, o GHCI podia inferir o tipo esperado de **`read`**.

Se usassemos ele como um booleano, ele saberia que deveria retornar um **`Bool`**. Mas agora ele só sabe que deve ser algum tipo da classe **`Read`**. Vamos dar uma olhada na declaração de tipo de **`read`**.

```
ghci> :t read
read :: (Read a) => String -> a
```

Viu? Ele retorna um tipo parte de **`Read`** mas como não usamos o resultado depois, ele não saberá qual tipo será. É por isso que podemos especificar explicitamente **type annotations** (anotações de tipos). *Anotações de tipos* servem para dizer qual tipo que você quer que uma expressão assuma. Fazemos isso adicionando **`::`** no fim da expressão com o tipo desejado. Observe:

```
ghci> read "5" :: Int
5
ghci> read "5" :: Float
5.0
ghci> (read "5" :: Float) * 4
20.0
ghci> read "[1,2,3,4]" :: [Int]
[1,2,3,4]
ghci> read "(3, 'a')" :: (Int, Char)
(3, 'a')
```

Na maioria das expressões, o compilador já pode assumir qual deve ser o tipo das expressões. Mas acontece dele não saber se deve ser **`Int`** ou **`Float`** para uma expressão como **`read "5"`**. Para ter certeza, Haskell deveria primeiro avaliar **`read "5"`**. Mas como Haskell é uma linguagem estaticamente tipada, precisa saber o tipo de todas as expressões na hora da compilação (ou no caso do GHCI, interpretação). Então dizemos ao Haskell: "Ei, essa expressão é desse tipo, caso não saiba!".

Os membros de `Enum` são tipos que possuem uma seqüência. A maior vantagem da typeclass **`Enum`** é poder ser usada em ranges de listas. Seus tipos têm sucessores e predecessores definidos, que podem ser conseguidos pelas funções **`succ`** e **`pred`**. Fazem parte dessa classe os tipos: **`()`**, **`Bool`**, **`Char`**, **`Ordering`**, **`Int`**, **`Integer`**, **`Float`** e **`Double`**.

```
ghci> ['a'..'e']
"abcde"
ghci> [LT .. GT]
[LT,EQ,GT]
ghci> [3 .. 5]
[3,4,5]
ghci> succ 'B'
'C'
```

`Bounded` são os tipos que possuem limites - máximo e mínimo.

```
ghci> minBound :: Int
-2147483648
ghci> maxBound :: Char
'\1114111'
ghci> maxBound :: Bool
True
ghci> minBound :: Bool
False
```

`minBound` e **`maxBound`** são diferenciados por ter tipo **`(Bounded a) => a`**. São constantes polimórficas.

Todas tuplas não-vazias também estão em **`Bounded`**.

```
ghci> maxBound :: (Bool, Int, Char)
(True,2147483647,'\1114111')
```

`Num` é uma typeclass numérica. Seus membros têm a função de agir como números. Vamos ver o tipo de um número.

```
ghci> :t 20
20 :: (Num t) => t
```

Parece que todos os números são constantes polimórficas. Elas podem tomas a forma de qualquer tipo da typeclass **`Num`**.

```
ghci> 20 :: Int
20
ghci> 20 :: Integer
20
ghci> 20 :: Float
20.0
ghci> 20 :: Double
20.0
```

Esses são os tipos da typeclass **`Num`**. Se verificar o tipo de **`*`**, descobrirá que ela aceita qualquer número.

```
ghci> :t (*)
(*) :: (Num a) => a -> a -> a
```

Recebe três números do mesmo tipo. É por isso que **`(5 :: Int) * (6 :: Integer)`** resultará em erro e **`5 * (6 :: Integer)`** funcionará e retornará um **`Integer`**, já que 5 pode tomar a forma de um **`Int`** ou de um **`Integer`**.

Para estar em **`Num`**, o tipo já deve estar em **`Show`** e **`Eq`**.

`Integral` também é uma typeclass numérica. Enquanto **`Num`** inclui todos os números (reais e inteiros), **`Integral`** apenas inteiros. Essa typeclass é composta por **`Int`** e **`Integer`**.

`Floating` inclui apenas números de ponto flutuante, então são **`Float`** e **`Double`**.

Uma função muito útil para lidar com números é `fromIntegral`. A declaração do seu tipo é **`fromIntegral :: (Num b, Integral a) => a -> b`**. Assim, vemos que ela recebe um número inteiro e transforma-o em algo mais genérico. Isso é útil quando você precisa que tipos inteiros e ponto flutuante trabalhem juntos. Por exemplo, a função **`length`** tem uma declaração de **`length :: [a] -> Int`** ao invés de ter algo mais geral como **`(Num b) => length :: [a] -> b`**. Acho que está assim por razões históricas, o que, na minha opinião, é besteira. Ainda assim, se tentarmos somar o tamanho de uma lista (**`length`**) com **`3.2`** teremos um erro, pois

não é possível somar um `Int` com um número de ponto flutuante. Então para contornar, `fromIntegral (length [1,2,3,4]) + 3.2` funciona perfeitamente.

Note que `fromIntegral` tem mais de um class constraint em sua declaração de tipo. Como pode ver, isso é válido, desde que estejam separados por vírgulas dentro de parênteses.

Começando Índice Sintaxe em Funções