Capítulo 3
Sistema de tipagem

Variáveis de tipo

Como foi visto nos capítulos anteriores, todas as operações em haskell, como soma, subtração e etc são funções. Sendo funções, elas podem possuir parâmetros, que, por sua vez, possuem tipos, que podem ou não influenciar no tipo de retorno dessas funções. Para averiguar estas minúcias, o GHCI dispõe do ":t <nomeFunção>" que fornece informações acerca dos tipos de entrada e de saída. Por exemplo, caso seja necessário averiguar a assinatura da função "+", basta usar ":t (+)" no ghci e, prontamente, ele a retornará.

```
1. :t (+)
        (+) :: Num a => a -> a -> a
```

Com esta assinatura, conseguimos observar algumas coisas novas, como a utilização de uma *variável de tipo*, neste caso a letra "a". Uma variável de tipo, ou *"type variable"* em inglês, é uma espécie de variável – como o próprio nome sugere – usado para representar, neste caso, o conjunto de todos os tipos numéricos. Portanto, a assinatura da função "+" diz basicamente que será aceito qualquer tipo "a" que possa ser entendido como um número. Observe outro exemplo :

```
1. :t snd
        fst :: (a, b) -> b
```

Esta assinatura nos diz que quando uma tupla composta por tipos "a" e "b", respectivamente, for fornecida a função "snd", ela sempre retornará o segundo elemento, neste caso "b". Perceba que a utilização de variáveis de tipo diferentes ("a" e "b") não expressa necessariamente que os tipos dos elementos que compõe a tupla *devem* ser diferentes, mas que eles *podem* ser diferentes. Funções que recebem variáveis de tipo são chamadas *funções polimórficas*.

Perceba uma coisa curiosa: enquanto o retorno de ":t (+)" possui a especificação que os parâmetros passados devem ser tipos numéricos (Num a =>), o retorno de ":t snd" não tem nenhuma restrição, o que significa que as tuplas suportam qualquer tipo de parâmetro. O símbolo que denota essa restrição é o "=>" e é responsável por denotar uma "restrição de classe", ou *"class constraint"* em inglês. No entanto, o que são essas "classes"?

NOTE : ao analisar a assinatura de uma função infixa usando o ":t", sempre será necessário pôr a função envolta por parênteses,

NOTE : Apesar de as variáveis de tipos dos exemplos serem todas letras, como "a" e "b", elas poderiam ter nomes mais auto-explicativos, como "parametro1" e "parametro2", por exemplo.

Typeclasses

Em haskell, existem espécies de “classes de tipos”, de tal forma que, caso tipos *x* e *y* pertençam ao mesmo conjunto *A* isso significa que esses tipos compartilham a característica principal da classe tipo A[1]. Ou seja, todos os tipos conhecidos no haskell se tratam de instâncias destas classes e é até mesmo possível que essas classes sejam instâncias de outras classes. Esses conjuntos são chamados “*typeclasses*” e o “Num” 1° exemplo da seção anterior é um deles. A seguir, os principais typeclasses do haskell são listados.

1. **Integral :** representa todos os tipos inteiros disponíveis no haskell, ou seja, o “Int” e o “Integer”. A única diferença entre o 1° e o 2° é que o primeiro possui limite de representação, enquando o segundo não. Algumas funções que só aceitam o Integral são pontuadas abaixo :

```
1. :t mod
        mod :: Integral a => a -> a -> a
2. :t div
        div :: Integral a => a -> a -> a
3. :t even --checa se um Integral é par
        even :: Integral a => a -> Bool
4. :t odd -- checa se um Integral é impar
        odd :: Integral a => a -> Bool
5. :t gcd -- calcula o MDC de 2 Integrals, a sigla vem do inglês “greatest common divisor”
        gcd :: Integral a => a -> a -> a
6. :t lcm --calcula o MDC de 2 Integrals, a sigla vem do inglês “least common multiple”
        lcm :: Integral a => a -> a -> a
```

2. **Floating :** representa todos os tipos de ponto flutuante disponíveis no haskell, ou seja, “Float” e “Double”. Observe algumas funções que aceitam Floating.

```
1. :t (**)
        (**) :: Floating a => a -> a -> a
2. :t sqrt -- acha a raiz quadrada de um número, o mnemônico vem do inglês “square root”
        sqrt :: Floating a => a -> a
3. :t sin -- computa o seno de um ângulo
        sin :: Floating a => a -> a
4. :t cos-- computa o cosseno de um ângulo
        cos :: Floating a => a -> a
5. :t tan -- calcula a tangente de um ângulo
        tan :: Floating a => a -> a
6. :t log – calcula o logaritmo natural de um número
        log :: Floating a => a -> a
```

3. Num : representa todos os números do haskell, por causa disto tanto “Integral”, quanto “Floating” pertencem a ela.

```
1. :t (+)
        (+) :: Num a => a -> a -> a
2. :t (-)
        (-) :: Num a => a -> a -> a
```

1 Que, por sinal, pode ser qualquer coisa, como ser um número inteiro menor que 10, por exemplo.

```
3. :t (*)
        (*) :: Num a => a -> a -> a
4. :t (^)
        (^) :: (Num a, Integral b) => a -> b -> a
```

4. **Eq :** representa os tipos que podem ser comparados através das funções “==” e “/=”. Portanto, o typeclass “Num” pertence a “Eq”, assim como os tipos caractere, string e, curiosamente, Bool. Além desses elementos, as funções “==” e “/=” também pertencem a Eq, como é de se esperar.

```
1. True == False
        False
2. True /= False
        True
3. :t (==)
        (==) :: Eq a => a -> a -> Bool
4. :t (/=)
        (/=) :: Eq a => a -> a -> Bool
```

5. **Ord:** Classe de tipo que agrupa todos os tipos que possuem uma relação de grandeza, por exemplo, ser maior ou menor a algo. Por incrível que pareça, o Ord é uma “subclasse” de Eq.

```
1. True > False
        True
2. False < True
        True
3. [True] > [False, False, False,False, False, False]
        True
4. :t (<)
        (<) :: Ord a => a -> a -> Bool
5. :t (<=)
        (<=) :: Ord a => a -> a -> Bool
```

6. **Show:** Possui todos os tipos que são passíveis de impressão na forma de uma string. A grande maioria, se não todos, os tipos e typeclasses apresentados até agora, são uma instância de Show. A função que é a marca registrada deste typeclass é um homólogo[2] dela : show. Esta função é muito utilizada para converter tipos em strings. Isto torna a impressão dos dados possível.

```
1. :t (show)
        show :: Show a => a -> String
2. show 12.5
        "12.5"
```

2 Neste contexto, “Homólogo” significa ter o mesmo nome.

```
3. show [True, True, True]
        "[True,True,True]"
4. show "tous, accourez"
        "\"tous, accourez\""
5. show [("Unity",3), ("Hapiness", 10), ("Charity", 14)]
        "[(\"Unity\",3),(\"Hapiness\",10),(\"Charity\",14)]"
```

7. **Read:** O Read é uma classe de tipo curiosa, pois conceitualmente falando, ela tem a característica singular de ter todas as strings que possam ser representadas como um tipo do haskell, como "subclasses" dela. De modo prática, enquanto a classe de tipo "Show" tem como "marca registrada" a função "show" para que seja possível a escrita de dados na forma de uma string, o "Read" tem como marca registrada a função "read", que torna possível a manipulação de tipos numéricos, booleanos e etc que foram representados como strings. Isto é feito através da conversão destes tipos. Observe os exemplos à baixo.

```
1. :t read
        read :: Read a => String -> a --recebe uma string e retorna um tipo que pertence à Read
2. read "3.1" :: Float
        3.1
3. read "12" + 2 -- O retorna da função "read" está sendo usado em uma expressão
        14
4. read "[1,3,7]" :: [Int]
        [1,3,7]
5. read "[1,3,7,12]" ++ [33, 144000]
        [1,3,7,12,33,144000]
6. read "[True, True, True]" :: [Bool]
        [True,True,True]
7. read "[True, True, True]" > [False, False, False, False, False, False]
        True
8. [read "1" :: Int .. read "14" :: Int]
        [1,2,3,4,5,6,7,8,9,10,11,12,13,14]
9. read "Peace" :: [Char] -- Não é possível converter uma string em uma string
        "*** Exception: Prelude.read: no parse
10. read "7.14"
        *** Exception: Prelude.read: no parse
```

Perceba uma coisa interessante: todas as vezes em que o retorno da função "read" ***não*** é usado em uma expressão maior[3], é necessário declarar explicitamente qual o tipo do parâmetro que "read" está recebendo através de um *anotação de tipo*[4]. Foi por causa da ausência desta especificação que a linha 7 gerou um erro. Observe também que o retorno de "read" estava sendo usado em uma expressão maior nas funções das linhas 2, 4 e 5 o que possibilitou com que o haskell conseguisse inferir qual era a conversão que se desejava fazer, logo, dispensando a necessidade de um *anotação de tipo*.

8. **Enum :** É a classe responsável por definir as operações permitidas nos tipos sequencialmente ordenados, portanto, todos os tipos que tenham :

3 Como nos exemplos das linhas 2, 4, 6 e 8.

4 Neste caso, o texto em vermelho. O termo se traduz para o inglês como *"type annotation"*

1. Predecessor
2. Sucessor
3. limite de representação superior
4. limite de representação inferior

São instâncias desta classe de tipo. Em virtude de ser uma classe de tipo mais conceitual que as demais, não há exemplos significativos de funções que a usem.

9. **Foldable** : Representa todos as coleções de dados cujos elementos podem ser “dobrados” em um único valor. Por exemplo, considere uma lista com inteiros, é possível “dobrar” os elementos da lista através de um somatório, então todos os seus elementos seriam “dobrados” em um único. Listas são instâncias de “Foldable”.A seguir, consta algumas funções que são capazes de performar esta combinação e que recebem dados do tipo Foldable.

```
1. :t foldr
        Foldable t => (a -> b -> b) -> b -> t a -> b
```

- Antes de tudo, é necessário entender a assinatura da função “foldr”. Portanto, observa-se que ela recebe três parâmetros, são eles :
    1. (a -> b -> b)
       Uma função que recebe duas variáveis “a” e “b” e retorna um valor “b”. Lembre-se que “a” e “b” são variáveis de tipo, portanto esta assinatura indica que a função recebe dois tipos que podem ou não serem iguais “a” e “b” e retorna um valor pertencente ao tipo do segundo parâmetro.
    2. b
       Um parâmetro “b”
    3. t a
       Esta notação indica que será passado um Foldable “t”, cujos elementos são de tipo “a”. No caso de “t” ser uma lista, você até pode compreender “t a” como sendo equivalente à “[a]”
- Por fim, a função retorna uma variável de tipo “b”

Feito estas explicações, afinal de contas esta função serve para quê ? Observe o exemplo abaixo para descobrir e entender melhor :

```
1. foldr (+) 0 [1..5] -- realiza o somatório de 1 à 5
        15
```

Conceitualmente, este código pode ser entendido como a soma recursiva entre os elementos em [1..5] e a variável que serve como acumulador, que neste exemplo é zero. Além disto, a soma é feita da direita para a esquerda (**r**ight to left).

Praticamente, o acumulador, que é passado no segundo parâmetro (neste caso zero) é somado com o último elemento de [1..5], feito isto o penúltimo elemento é somado com a soma entre o acumulador e o último elemento e assim recursivamente, até que todos estas operações tenham sido aplicadas para todos os elementos de [1..5]. Abaixo, há o somatório relativo a como as chamadas recursivas aconteceram.

1.(0+5)
2. 4+(0+5)
3. 3+(4+(0+5))
4. 2+(3+(4+(0+5)))
5. 1+(2+(3+(4+(0+5))))

Observe que o somatório foi feito da **direita para a esquerda.**

Caso a função "foldl" fosse aplicada, o somatório aconteceria da **esquerda para a direita**. Observe :

```
1. foldl (+) 0 [1..5] -- realiza o somatório de 1 à 5
        15
```

Esta função somará o acumulador com o primeiro elemento, depois realizará a soma entre o segundo elemento e a soma entre o acumulador com o primeiro elemento, até que esta operação tenha sido aplicada recursivamente para todos os elementos de [1..5]

Observe a estrutura do somatório das chamadas recursivas dessa função:

1.(0+1)
2. 2+(0+1)
3. 3+(2+(0+1))
4. 4+(3+(2+(0+1)))
5. 5+(4+(3+(2+(0+1))))

```
1. :t elem
        elem :: (Foldable t, Eq a) => a -> t a -> Bool
```

Primeiramente, perceba que para realizar mais de uma restrição de classe, é necessário pôr as variáveis de tipo que estão sendo declaradas entre parênteses, logo, ao fazer "(Foldable t, Eq a)", é declarado uma variável de tipo "t" da classe "Foldable" e uma "a" da classe "Eq". Dito isto, perceba que é necessário que "a" seja uma instância de Eq, pois internamente a função percorre todo o "Foldable" realizando a operação de igualdade para checar se "*a*" está contido em "t".

10. **Bounded:** Representa os tipos que possuem limites de representação, alguns tipos que pertencem a este typeclass são : Int, Char, Bool, dentre outros.

```
1. :t minBound -- retorna o limite inferior
        minBound :: Bounded a => a
2. :t maxBound – retorna o limite superior
        maxBound :: Bounded a => a
3. minBound :: Int
        -9223372036854775808
4. maxBound ::Int
        9223372036854775807
```

```
5. minBound :: Bool
        False
6. maxBound :: Bool
        True
7. minBound :: Char
        '\NUL'
8. maxBound :: Char
        '\1114111'
9. maxBound :: (Bool, Char, Int)
        (True,'\1114111',9223372036854775807)
```

11. **Ordering:** Ela é uma classe de tipo que, de fato, representa valores. Eles são 3 :
    1. LT(*Less than)*: Menor quê
    2. EQ(*Equal):* Igual
    3. GT(*Greater than)*: Maior quê

```
5. :t compare
        compare :: Ord a => a -> a -> Ordering
6. compare 7 6 -- sete é maior que 6, logo retorna GT
        GT
7. compare 7 7
        Eq
8. compare 6 7
        LT
```

NOTE : Para um tipo pertencer a “Num”, ele antes precisa ser instância de Show e Read

NOTE : Tudo o que vem antes do símbolo “=>” se chama “*class constraint”*, ou restrição de classe.

NOTE : Para obter mais informações acerca das classes de tipo, basta inserir o comando “:info <classeTipo>” no ghci.

Anotações de tipo

Uma anotação de tipo é um recurso disponível no haskell que permite especificar o tipo de uma expressão, isto ajuda a evitar ambiguidades. Ela foi usada algumas vezes durante a seção anterior.

```
1. 27 :: Float
        27.0
{-Alguns exemplos da seção anterior com anotações de tipo-}
2. read "3.1" :: Float
3. read "[1,3,7]" :: [Int]
```