Haskell:

Polimorfismo e Classes

Profa. Dra. Gina M. B. Oliveira

Função Monomórfica

Função monomórfica: é definida apenas para tipos específicos.

Exemplos:

1) Função sucessor (definida para inteiros)

```
sucessor :: Int -> Int
sucessor x = x+1

Main> sucessor 5
6
```

Função Monomórfica

2) Função concatena (definida para lista de strings)

```
concatena::[String]->[String]->[String]
concatena [ ] y = y
concatena (x:xs) y = x: concatena xs y

Main> concatena ["aba", "ba"] ["va", "vav"]
["aba","ba","va","vav"]
```

Se tentarmos aplicar a definição acima para as listas de inteiros, acarretará um erro.

```
Main> concatena [1,2,3] [4,5]
error
```

Função Polimorfa

Entretanto, podemos definir uma função concatena mais genérica, que se aplique a
todos os tipos de lista.

A linguagem Haskell permite a definição de funções para diferentes tipos.

Função polimorfa: uma mesma definição pode ser usada para diversos tipos de dados.

Por exemplo, nas funções do Prelude padrão existem diversas funções genéricas sobre listas,
que se aplicam a listas de qualquer tipo. Ex: head, tail, ++, etc. Outro exemplo são as funções de tuplas,
que se aplicam a qualquer tipo de tupla. Por exemplo: fst, snd.

Dizemos que essas funções são polimorfas (várias formas).

Mas como definimos funções polimorfas?

Função Polimorfa

Basta utilizarmos variáveis na definição dos tipos das funções.

Por exemplo, os tipos das seguintes funções são assim definidos:

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

tail :: [a] -> [a]

(++) :: [a] -> [a] -> [a]

fst :: (a, b) -> a

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

Observem que nas funções de tuplas uma segunda variável foi necessária, uma vez que os tipos das tuplas podem ser
diferentes (ou seja, a e b podem ser do mesmo tipo, mas também podem não ser.

Função Polimorfa

Dessa forma, podemos reescrever nossa função de concatenação para ser mais genérica:

Ex: Função concatena (definida para listas qualquer tipo)

```
concatena::[a]->[a]->[a]
concatena [ ] y = y
concatena (x:xs) y = x: concatena xs y
```

Agora podemos aplicá-la a diferentes tipos de listas, não apenas listas de strings:

```
Main> concatena [1,2,3] [4,5]
[1,2,3,4,5]
Main> concatena ['a','b'] ['c']
"abc"
Main> concatena ["aba", "ba"] ["va", "vav"]
["aba","ba","va","vav"]
```

Observe que a função é válida mesmo para tipos estruturados (listas e tuplas):

```
Main> concatena [('a',1),('b',2)] [('c',3)]
[('a',1),('b',2),('c',3)]
```

Classes de Tipos

Agora, vejamos a função sucessor. Poderíamos redefini-la como:

sucessor :: a-> a

Veja que, para alguns tipos (no caso os numéricos Int, Integer, Float e Double), essa função é adequada, pois é possível aplicar a operação "+1". Mas para outros (por exemplo, Char, String, [Int], (Int,Float)) não é possível aplicar a operação.

A linguagem Haskell também nos permite definir que uma função pode ser usada para vários (alguns) tipos de dados, mas não para todos os tipos.

Nesse caso, usamos o conceito de Classe para definirmos uma restrição de tipo.

Classes de Tipos: são usadas para permitir diferentes tipos para uma função ou operação.

Classes de Tipos

• **Classe**: coleção de tipos para os quais uma função está definida.

Existem classes já definidas no Haskell: Num, Eq, Ord, Enum, etc.

Por exemplo, a classe Num agrupa os tipos numéricos: Int, Integer, Float e Double.

Dizemos que os tipos Int, Integer, Float e Double são instâncias da classe Num.

Quando fazemos a definição do tipo de uma função, podemos utilizar uma restrição de tipo para uma ou mais variáveis genéricas utilizadas.

No exemplo anterior:

```
sucessor :: Num a => a ->  a
sucessor x = x+1
```

A restrição de tipo para a variável <a> é dada por Num a, ou seja, <a> deve ser de um tipo que pertença à classe Num.
A parte anterior ao símbolo => é chamada de contexto.

Classe Num

```
sucessor :: Num a => a ->  a
sucessor x = x+1
```

Com essa definição de tipo, a função sucessor pode ser aplicada a qualquer tipo numérico:

```
Main> sucessor 5
6

Main> sucessor 5.2
6.2

Main> sucessor 'a'
error
```

Classe Num

Se avaliarmos os tipos de outras funções/operadores do Prelude veremos que elas usam uma restrição de tipo para a classe numérica.

Ex:

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

Podemos definir outras funções exclusivamente numéricas usando a Classe Num:

```
somatoria :: Num a => [a] -> a
somatoria [ ] = 0
somatoria (x:xs) = x + somatoria xs

produtorio :: Num a => [a] -> a
produtorio [ ] = 1
produtorio (x:xs) = x * produtorio xs
```

Classes Num

As restrições de tipo podem ser aplicadas a variáveis selecionadas:

Ex: Funcao f

```
f:: Num a => a -> [b] -> [b] ->(a,[b])
f x y z = (x + 1, y ++ z)

Main> f 1 [2] [3]
(2,[2,3])

Main> f 1 2 3
error

Main> f 5.2 ['a'] ['b']
(6.2,"ab")
```

A Funcao f recebe 3 argumentos x, y e z, sendo o 1o numérico e o 2o e o 3o sao listas.
Retorna uma dupla onde o 1o elemento é x+1 e o segundo é a lista resultante de y++z

Classe Eq

As classes Eq (Igualdade) e Ord (Ordem) também são muito utilizadas em Haskell.

Essas classes são muito utilizadas em funções que envolvem comparações entre elementos de um mesmo tipo.

A classe igualdade (Eq) compreende todos os tipos para os quais a operação (==) está definida.

São instâncias de Eq os tipos primitivos e as listas e tuplas de instâncias de Eq

Ex: Int, Float, Char, Bool, [Int], (Int,Bool),[[Char]], [(Int,[Bool])]

Exemplos:

Classes Eq

1) Função iguais3

```
iguais3 :: Eq t => t -> t -> t -> Bool
iguais3 n m p = (n == m) && (m == p)

Main> iguais3 1 2 3
False

Main> iguais3 'c' 'c' 'c'
True

Main> iguais3 1 "um" "um"
error

Main> iguais3 ('a',2) ('a',2) ('a',2)
True

Main> iguais3 [1,2,3] [1,2,3] [1,2,3]
True
```

Classes Eq

2) Função member

```
member ::Eq t => t -> [t] -> Bool
member _ [ ] = False
member a (x:xs) = if (a==x) then True else member a xs
```

```
Main> member 1 [2,3]
False

Main> member 'c' ['a','b','c','d']
True

Main> member ('a',2) [('b',4),('a',2)]
True

Main> member [1,2,3] [[1,2],[3]]
False

Main> member 2 ['a,'b','c','d']
error
```

Classes Ord

A classe ordem (Ord) compreende todos os tipos para os quais as operações (==), (<), (>), (<=), (>=), max, min estão definidas. Ou seja, ela compreende a operação (==), que também define a única rerstrição de Eq.
Por isso, Eq é uma superclasse de Ord.

Exemplos:

1) Função antes

```
antes:: Ord a => a->a->Bool
antes a b = if a < b then True else False

Main> antes 1 2
True
Main> antes 1 1
False
Main> antes 2 1
False
Main> antes 'a' 'c'
True
Mais> antes "a" "ac"
True
```

Classes Ord

2) Funcao Ordenada: verifica se os elementos de uma lista estão ordenados

```
ordenada:: Ord a => [a] ->Bool
ordenada (a:b:[]) = if a < b then True else False
ordenada (a:b:resto) = if a < b then ordenada (b:resto) else False

Main> ordenada [1,2,3,4,5]
True
Main> ordenada [1,2,6,4,5]
False
Main> ordenada ['a', 'c', 'z']
True
Main> ordenada ["a", "ac","ca"]
True
Main> ordenada ["a","ca", "ac"]
False
Main> ordenada [("a",2),("ca",4), ("ca",5)]
True
Main> ordenada [("a",2),("ca",4), ("ca",3)]
False
Main> ordenada [("a",2),("ca,'a'), ("ca",3)]
error
```

Outras classes

Existem outras classes pré-definidas em Haskell.

Por exemplo:

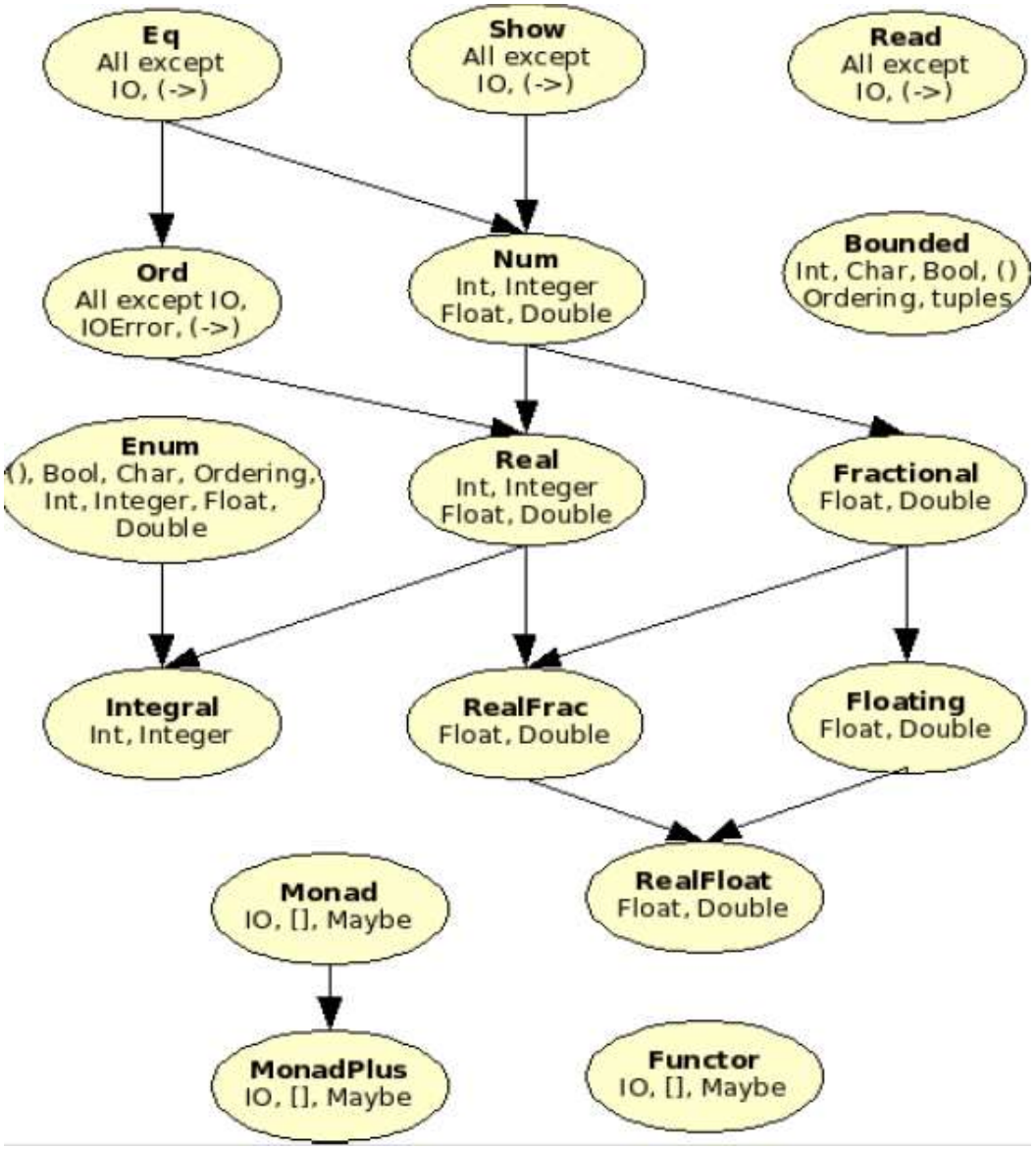
Enum: enumeração (variáveis que podem ser enumeradas)

Show: apresentação (variáveis que podem ser transformadas em Strings)

Fractional: números que podem sofrer a divisão não inteira (Float e Double)

Integral: podem ser submetidas à divisão inteira (Int e Integer)

Outras classes



Definição de classes

É possível ao usuário definir suas próprias classes, através da declaração **class**.

Também é possível redifinir uma classe já pré-definida no Prelude
(sobrecarregamento: mudar a definição padrão)

Esse ponto foge ao escopo desse material. Entretanto, segue a definição da classe Eq como exemplo de definição de classes:

```
class Eq t where
   (==) :: t -> t -> Bool
```

Define-se o que é necessário para um tipo t ser da classe Eq:
deve possuir uma função (==) definida sobre t,
do tipo t -> t -> Bool