

Programação Genérica e Classes de Tipos

Programação Funcional

Prof. Maycon Amaro

Polimorfismo

- ▶ Significa “de muitas formas”
- ▶ Na computação, descreve funções que podem ser usadas por vários tipos diferentes
- ▶ O contrário seria **monomorfismo**

Exemplo de Monomorfismo

```
int retorna_primeiro_int(int[] v) {  
    return v[0];  
}
```

```
double retorna_primeiro_double(double[] v) {  
    return v[0];  
}
```

```
char retorna_primeiro_char(char[] v){  
    ...  
}
```

- ▶ O comportamento de retornar o primeiro elemento de um vetor independe do tipo dos elementos.
- ▶ Ter que escrever uma função separada para cada tipo é improdutivo
- ▶ Com polimorfismo, uma única implementação é suficiente.

Em C/C++ o uso de templates permite aplicar polimorfismo ao nosso exemplo.

```
template<typename T>
T retorna_primeiro(T[] v) {
    return v[0];
}
```

Essa função pode ser usada com vetores de qualquer tipo.

Tipos de Polimorfismo

Podemos dividir em três categorias

- ▶ Polimorfismo paramétrico
- ▶ Polimorfismo restringido (ad-hoc)
- ▶ Polimorfismo de inclusão

Polimorfismo Paramétrico

Os tipos envolvidos são substituídos por um símbolo coringa, que podem se referir à qualquer tipo.

```
template<T>
T retorna_primeiro(T[] v) {
    return v[0];
}
```

Em Haskell, basta substituir o tipo por uma letra minúscula

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

primeiro está no prelúdio como head

Polimorfismo de Inclusão

Acontece por meio de *subtyping*, em que tipos podem ser hierarquizados. Comum na Programação Orientada à Objetos, através do conceito de **herança**.

Assim, funções definidos para o supertipo (o tipo “pai”) podem ser usados pelos subtipos (os tipos “filho”).

Polimorfismo Restringido

Define uma interface em comum para tipos específicos. No polimorfismo de sobrecarga, duas funções para tipos diferentes podem ter o mesmo nome.

```
int retorna_primeiro(int[] v) { }
```

```
double retorna_primeiro(double[] v) { }
```

```
char retorna_primeiro(char[] v) { }
```

Isso não é permitido em Haskell.

- ▶ O polimorfismo restringido aceito por Haskell consiste em estabelecer restrições sobre os tipos genéricos.
- ▶ Essas restrições acontecem por meio de **classes de tipos**.

```
minhaFuncao :: Num a => a -> a -> a  
minhaFuncao x y = 2 * x + y
```

Classes de Tipos

- ▶ Descreve como um conjunto de tipos é *consumido* ou usado em computações
- ▶ Permitem generalizar algumas operações para um conjunto de tipos
- ▶ Podemos definir nossas próprias classes, mas veremos primeiro as principais classes do Prelúdio.

A Classe Eq

- ▶ Alguns tipos de dados não possuem uma forma precisa de definir igualdade
- ▶ Ao invés de permitir que todo tipo possa ser comparado, Haskell define a classe Eq
- ▶ Descreve como comparar os elementos daquele tipo
- ▶ Os tipos Int, Float, Double, Char, String, Bool, etc., são todos instâncias da classe Eq.

```
ghci> :info Eq
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    {-# MINIMAL (==) | (/=) #-}
```

Definindo uma Instância

```
data Sorvete = Flocos | Chocolate | Morango
```

```
instance Eq Sorvete where
```

```
    (==) Flocos    Flocos    = True
```

```
    (==) Chocolate Chocolate = True
```

```
    (==) Morango   Morango   = True
```

```
    (==) _         _         = False
```

Deixando o compilador defini-la

```
data Sorvete = Flocos | Chocolate | Morango  
  deriving (Eq)
```

A Classe Ord

Define operações para pôr elementos em ordem.

```
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  max :: a -> a -> a
  min :: a -> a -> a
  {-# MINIMAL compare | (<=) #-}
```


Exemplo

```
data Numero = Tres | Um | Dois | Quatro
  deriving (Eq)
```

```
instance Ord Numero where
  (<=) Um      _      = True
  (<=) Quatro Quatro = True
  (<=) Quatro _      = False
  (<=) Dois   Um      = False
  (<=) Dois   _      = True
  (<=) Tres   Um      = False
  (<=) Tres   Dois    = False
  (<=) Tres   _      = True
```

Deixando o compilador definir

```
data Numero = Um | Dois | Tres | Quatro  
  deriving (Eq, Ord)
```

Para usar a função `sort` do prelúdio, o tipo da lista deve ser uma instância de `Ord`

```
ghci> import Data.List
ghci> :t sort
sort :: Ord a => [a] -> [a]
```

A Classe Show

Define a operação de converter em String. Um tipo instância de Show é um que pode ser exibido em texto *human-readable*. Não é adequada para serialização.

```
class Show a where
  showsPrec :: Int -> a -> ShowS
  show      :: a -> String
  showList :: [a] -> ShowS
  {-# MINIMAL showsPrec | show #-}
```

Exemplo

```
data Sorvete = Flocos | Chocolate | Morango
```

```
instance Show Sorvete where  
  show Flocos      = "Flocos"  
  show Chocolate   = "Chocolate"  
  show Morango     = "Morango"
```

versus

```
data Sorvete = Flocos | Chocolate | Morango  
  deriving (Show)
```

A Classe Num

Define operações numéricas. Não pode ser derivada automaticamente.

```
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
  {-# MINIMAL (+), (*), abs, signum, fromInteger,
        (negate | (-)) #-}
```

A Classe Real

Define operações sobre números reais.

```
class (Num a, Ord a) => Real a where  
  toRational :: a -> Rational  
  {-# MINIMAL toRational #-}
```

Outras classes numéricas

- ▶ Fractional
- ▶ Integral
- ▶ RealFrac

No GHCi, utilize `:info` para obter informações sobre elas.

A Classe Enum

Define operações para enumeração.

```
class Enum a where
  succ :: a -> a
  pred :: a -> a
  toEnum :: Int -> a
  fromEnum :: a -> Int
  enumFrom :: a -> [a]
  enumFromThen :: a -> a -> [a]
  enumFromTo :: a -> a -> [a]
  enumFromThenTo :: a -> a -> a -> [a]
  {-# MINIMAL toEnum, fromEnum #-}
```

Algumas dessas operações retornam *listas infinitas*.

Definindo uma Classe de Tipos

```
data Bit = Zero | Um

class Binary a where
  toBin :: a -> [Bit]
  fromBin :: [Bit] -> a
  {-# MINIMAL toBin, fromBin #-}

instance Binary Int where
  toBin x = -- conversao decimal -> binario
```

Exemplos de Implementações

Insertion Sort

Só faz sentido com listas de elementos que podem ser postos em ordem.

```
insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) | x < y      = x:y:ys
                  | otherwise = y:(insert x ys)
```

```
isort :: Ord a => [a] -> [a]
isort [] = []
isort (x:xs) = insert x (isort xs)
```

Árvore Binária de Pesquisa

Para satisfazer as restrições, os elementos precisam ser comparáveis.

```
data Ord a => BinTree a
  = Empty
  | Node (BinTree a) a (BinTree a)
  deriving (Eq, Show)
```

Conversor de Temperaturas

As temperaturas podem ser qualquer número fracionário.

```
celsiusToFahrenheit :: Fractional => a -> a  
celsiusToFahrenheit x = 9 / 5 * x + 32
```

Em Outras Linguagens

C++20

A biblioteca `concepts` do padrão C++20 fornecem funcionalidades semelhantes às classes de tipos.

```
template<typename T>  
requires std::totally_ordered<T>  
void quick_sort(std::vector<T> v) { ... }
```


Rust

As Traits de Rust funcionam como classes de tipos.

```
impl<T: Display> ToString for T {  
    // --snip--  
}
```

Dúvidas?