Paradigmas de Programação

Prof. Maicon R. Zatelli

Haskell - Programação Funcional
Classes (Type Classes)

Universidade Federal de Santa Catarina
Florianópolis - Brasil
2018/1

Haskell

Classes em Haskell definem métodos ou conjunto de operações sobre tipos, assim uma classe corresponde a um conjunto de tipos que possuem certos métodos (ou operações).

- É por meio de classes que o Haskell lida com polimorfismo ad-hoc de sobrecarga.
- Porém, note que em Haskell não há a noção de objeto, visto que não há a noção de estado.
- As principais classes já predefinidas no módulo Prelude são: `Eq`, `Ord`, `Enum`, `Read`, `Show`, `Functor`, `Monad` e `Bounded`
- Outras classes numéricas são `Num`, `Integral`, `Real`, `Fractional`, `Floating`, `RealFrac` e `RealFloat`

Haskell

Haskell utiliza o conceito de classes para lidar com polimorfismo.

Polimorfismo paramétrico: são os tipos genéricos.

```
gerarPares :: [t] -> [u] -> [(t,u)]
gerarPares l1 l2 = [(a,b) | a <- l1, b <- l2]
```

- **t** e **u** podem ser qualquer tipo compatível com as operações realizadas. Neste caso, apenas estamos formando pares, portanto qualquer tipo é compatível com esta operação.

Haskell

Haskell utiliza o conceito de classes para lidar com polimorfismo.

Polimorfismo ad-hoc (coerção/coercion): tipos são convertidos de maneira implícita (ex: inteiros e reais, na operação de soma).

```
soma :: Float -> Float -> Float
soma x y = x + y + 1

main = do
    print (soma 21 10)
```

Haskell

Haskell utiliza o conceito de classes para lidar com polimorfismo.

Polimorfismo ad-hoc (sobrecarga/overloading): são os métodos que possuem o mesmo nome, mas variam no tipo e/ou quantidade de parâmetros e/ou retorno. Haskell não suporta este tipo de polimorfismo ad-hoc.

Tente executar...

```
soma :: Float -> Float -> Float
soma x y = x + y

soma :: Int -> Int -> Int
soma x y = x + y

main = do
    print (soma 21 10)
```

- O que acontece?

Haskell

Haskell utiliza o conceito de classes para lidar com polimorfismo.

Polimorfismo ad-hoc (sobrecarga/overloading): são os métodos que possuem o mesmo nome, mas variam no tipo e/ou quantidade de parâmetros e/ou retorno. Haskell não suporta este tipo de polimorfismo ad-hoc.

Tente executar...

```
soma :: Float -> Float -> Float
soma x y = x + y

soma :: Int -> Int -> Int
soma x y = x + y

main = do
    print (soma 21 10)
```

- O que acontece?

Mesmo assim, não é possível simplesmente usar um número de parâmetros diferentes para um mesmo nome de função.

Exemplo de classe

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

Aqui definimos um exemplo de class com o nome **Eq**, a qual diz que um tipo **a** é uma instância da classe **Eq**, somente se ela possui suporte ao operador ==. Em outras palavras, qualquer tipo **a** que deseja ser uma instância de **Eq** deve definir a operação ==. Por exemplo, para Int ser uma instância de **Eq**, temos que definir a operação `(==) :: Int -> Int -> Bool`.

Exemplo de instância

```
instance Eq Integer where
  x == y =  x `integerEq` y
```

Aqui definimos uma instância da classe **Eq**, sendo que a `= Integer`. Assim, o tipo **Integer** é agora uma instância da classe **Eq**, e `x == y = x `integerEq` y` é o método correspondente ao operador ==.

Exemplo de classe

```
class  Eq a  where
  (==), (/=) :: a -> a -> Bool
  x /= y =  not (x == y)
```

Aqui implementamos o método `x /= y = not (x == y)` correspondente ao operador $/=$

Exemplo de classe (herança)

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

- Eq é uma superclasse de Ord (Ord é uma subclasse de Eq).
- Qualquer tipo que é instância de Ord deve ser também uma instância de Eq.
- Ord é uma classe relativa a tipos de dados totalmente ordenáveis

Exemplo de classe (herança múltipla)

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

- Real é uma classe que é subclasse de Num e Ord
- Ela possui um método toRational que recebe um valor e retorna um número racional

Haskell

```
class (Num a) => MeuFloat a where
    (+++) :: a -> a -> a
    (***) :: a -> a -> a

    x *** y = x * x * y

instance MeuFloat Double where
    x +++ y = 2 * x + y

instance MeuFloat Integer where
    x +++ y = 10 * x + y
```

- Criamos uma classe `MeuFloat` com novas operações $+++$ e $***$
- O método padrão para $***$ está definido na classe `MeuFloat`
- O método para $+++$ está definido nas instâncias de `MeuFloat` (sobre os tipos `Double` e `Integer`)

Haskell

```
main = do
    print ((4::Integer) *** (2::Integer))
    print ((4.2::Double) *** (2::Double))
    print ((2::Integer) +++ (4::Integer))
    print ((2.0::Double) +++ (4.0::Double))
```

Saída

```
32
35.28
24
8.0
```

Haskell

```
class (Integral x) => MeuInt x where
    bigger  :: x -> x -> x
    smaller  :: x -> x -> x

    bigger a b | a > b = a
               | otherwise = b

    smaller a b | a == (bigger a b) = b
                | otherwise = a

instance MeuInt Integer
instance MeuInt Int
```

- Criamos uma classe `MeuInt` com novos métodos: *bigger* e *smaller*
- Os métodos para os dois novos métodos estão escritos na mesma classe
- Por fim, criamos uma instância de MeuInt com o tipo Integer

Haskell

```
main = do
    print (bigger (4::Integer) (12::Integer))
    print (smaller (4::Integer) (12::Integer))
```

Saída

```
12
4
```

Tipos de dados

Booleanos (`Bool`), Caracteres (`Char`, `String`), Numéricos (`Int`, `Integer`, `Float` e `Double`), algébricos (n-uplas) e abstratos (funções, `Maybe`, `Functor` etc.).

- Bool: False | True
- Char: 16 bits de representação Unicode
- String: equivale a [Char] e o contrutor nulo é uma n-upla vazia, representado por ().
- Int: inteiros de -2^{29} a $2^{29} - 1$
- Integer: inteiros de precisao arbitrária (até possuir espaço de memória)
- Float: número real de precisão simples (32 bits)
- Double: número real de precisão dupla (64 bits)

Tipos de dados

- Algébricos: coleção de valores organizados, tais como vetores, matrizes, duplas, triplas, n-uplas (ordenadas e não ordenadas).
- Listas: tipos algébricos formados de 2 construtores : e [ ].
  - Ex: 1:2:3:[] resulta em [1,2,3]
- Tuplas: tipos algébricos formados de 2 construtores , e ( ).
  - Ex: (,) 4 5 resulta em (4,5)

Tipos de dados

Funções, `Maybe`, `Either`, `Ordering`, `Functor` e ( ) são tipos abstratos de dados.

- Um tipo abstrato de dados é um tipo ou classe cujo comportamento é definido por um conjunto de valores e um conjunto de operações.
- Em um tipo abstrato de dados, há somente mensão sobre quais operações são realizadas, mas não como elas serão implementadas.

```
data Maybe a = Nothing | Just a
  deriving (Eq, Ord, Read, Show)
```

- Neste exemplo, o tipo de dados para Maybe está em aberto. É deixado para o usuário informar. É o **a**.

Leia mais: `https://wiki.haskell.org/Abstract_data_type`

Haskell - Alguns Links Úteis

- https://www.haskell.org/tutorial/classes.html
- https://wiki.haskell.org/Abstract_data_type
- https://en.wikibooks.org/wiki/Haskell/Classes_and_types

Haskell

Ver atividade no Moodle