Classes de tipos

Programação Funcional

Prof. Rodrigo Ribeiro

Setup

```haskell
{-# LANGUAGE InstanceSigs #-}
module Aula10 where

main :: IO ()
main = return ()
```

Anteriormente...

- Polimorfismo paramérico: funções funcionam de maneira uniforme para todos os tipos.

```
reverse :: [a] -> [a]
fst :: (a,b) -> a
```

Anteriormente...

- Sobrecarga: funções que operam de forma diferente sobre diferentes tipos.
  - Não são definidas para todos os tipos.
- Exemplos:

```
elem :: Eq a => a -> [a] -> Bool
(+)  :: Num a => a -> a -> a
```

Classes de tipos

- Nome inicia com letra maiúscula.
- Define uma variável de tipo que representa o tipo a ser sobrecarregado.
- Cada classe define um ou mais funções que devem ser implementadas por suas instâncias.

Exemplo

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

Instâncias

- Similares a declaração de classes.
- Variáveis são substituída por tipos concretos.
- Assinaturas de tipos devem ser substituídas por implementações das funções para o tipo em questão.

Exemplo

```
instance Eq Bool where
    True == True = True
    False == False = True
    _ == _ = False

    x /= y = not (x == y)
```

Exemplo

```haskell
data Point = Point Int Int

instance Eq Point where
    (Point x y) == (Point x' y') =
        x == x' && y == y'
    x /= y = not (x == y)
```

Exemplo

- Só podemos usar uma função sobrecarregada se essa possuir implementação para um certo tipo.

```
data Vec3 = Vec3 Int Int Int

Prelude*> (Vec3 0 1 1) == (Vec3 0 1 1)
No instance for (Eq Vec3)
```

Assinaturas em instâncias

- Por padrão, não podemos anotar tipos em instâncias.
- A extensão `InstanceSigs` habilita essa possibilidade.

Exemplo

```haskell
data BoolPair = BoolPair Bool Bool

instance Eq BoolPair where
   (==) :: BoolPair -> BoolPair -> Bool
   (BoolPair b1 b2) == (BoolPair b1' b2')
      = b1 == b1' && b2 == b2'
   (/=) :: BoolPair -> BoolPair -> Bool
   b1 /= b2 = not (b1 == b2)
```

Instâncias recursivas

- Instâncias sobre tipos polimórficos podem depender da existência de instâncias sobre seus parâmetros.

Exemplo

```
instance Eq a => Eq [a] where
    [] == [] = True
    (x : xs) == (y : ys)
       = x == y && xs == ys
    _ == _ = False

    xs /= ys = not (xs == ys)
```

Superclasses

- Classes podem exigir que todas suas instâncias possuam instâncias de outras classes.
- Nesse caso, dizemos que a primeira é uma subclasse da segunda. Ex: Ord é subclasse de Eq.

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

Significado de =>

- Em um tipo, restringe um tipo polimórfico.

```
elem :: Eq a => a -> [a] -> Bool
```

- Tipo a deve ser uma instância de `Eq`.

Significado de =>

- Em classes, define uma relação de subclasses.

```
class Eq a => Ord a where
```

- Todo tipo instância de `Ord` deve possuir uma instância de `Eq`

Significado de =>

- Em uma instância, define um requisito para a instância definida.

```
instance Eq a => Eq [a] where
```

- Especifica que para usar a igualdade sobre listas, devemos possuir a igualdade sobre elementos.

Definições default

- Diversas instâncias de `Eq` possuem a mesma implementação de `(/=)`:

```
x /= y = not (x == y)
```

- Tais definições padrão podem ser incluídas na declaração da classe.

Exemplo

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

    x /= y = not (x == y)
    x == y = not (x /= y)
```

Derivação automática

- O compilador de Haskell é capaz de produzir, automaticamente, definições de algumas funções sobrecarregadas.
- Exemplos: Eq, Ord, Show, Enum, etc. . .

Derivação automática

- A declaração seguinte gera implementações padrão de instâncias de Eq, Ord e Show para o tipo `Point2D`:

```
data Point2D
   = Point2D Int Int
     deriving (Eq, Ord, Show)
```

Definindo uma classe

- Operação para realizar scaling de formas e vetores.

```
data Vector
   = Vector Float Float
     deriving (Eq, Ord, Show)

data Shape
   = Rectangle Vector Float Float
   | Circle Vector Float
   deriving (Eq, Ord, Show)
```

Definindo uma classe

- Operação para realizar scaling de formas e vetores.

```
class Scale a where
   scale :: Float -> a -> a
```

Definindo instâncias

- Instância para Vector.

```
instance Scale Vector where
   scale s v@(Vector x y)
      = Vector (x * s') (y * s')
        where
         s' = s / norm v
         norm (Vector a b) = sqrt (a^2 + b^2)
```

Definindo instâncias

- Instâncias para Shape.

```
instance Scale Shape where
   scale s (Circle p r) = Circle p (s * r)
   scale s (Rectangle p w h)
     = Rectangle p (s * w) (s * h)
```

Funções sobre Scalable

- Podemos definir código que funciona para qualquer tipo Scalable.

```
double :: Scale a => a -> a
double s = scale 2.0 s
```

Mais instâncias

- Podemos definir instâncias sobre coleções.

```
instance Scale a => Scale [a] where
   scale s = map (scale s)
```

Tipos númericos

- Constantes numéricas são sobrecarregadas.

```
Prelude*>:t 3
3 :: Num a => a
```

Tipos numéricos

- Como 3 é sobrecarregado, podemos usá-lo como qualquer tipo instância de `Num`:

```
Prelude*> :t 3 :: Int
3 :: Int

Prelude*> :t 3 :: Float
3 :: Float

Prelude*> :t 3 :: Rational
3 % 1 :: Rational
```

Tipos Numéricos

- Definidos pela classe Num:

```
class Num a where
   fromInteger :: Integer -> a
   (+), (*), (-) :: a -> a -> a
   abs, signum :: a -> a
```

Tipos Numéricos

- Sobrecarga de valores numéricos é feita pela função `fromInteger`.
- Isto é, 3 :: Int é, na verdade, `fromInteger 3`.
- Isso permite a sobrecarga de valores quaisquer como números e expressões numéricas.

Exemplo

- Expressões aritméticas

```
data Exp
  = Const Int
  | Exp :+: Exp
  | Exp :-: Exp
  | Exp :*: Exp
  | Abs Exp
  | Sign Exp
  deriving (Eq, Ord, Show)
```

Exemplo

```
ex :: Exp
ex = Const 1 :+: Const 1
```

Exemplo

- instância de `Num`:

```
instance Num Exp where
  fromInteger = Const . fromInteger
  (+) = (:+:)
  (-) = (:-:)
  (*) = (:*:)
  abs = Abs
  signum = Sign
```

Exemplo

- Podemos usar a instância de `Num` para escrever expressões usando operadores de Haskell.

```
ex1 :: Exp
ex1 = 1 + 1
```

```
*Aula10> ex1
Const 1 :+: Const 1
```

Intervalos

- Sintaxe de intervalos usada em listas é, na verdade, uma operação sobrecarregada.
- A lista `[n..m]` é uma abreviação para `enumFromTo n m`.

```
enumFromTo :: Enum a => a -> a -> [a]
```

Essa função é definida na classe `Enum`

Intervalos

```
enumFrom :: Enum a => a -> [a]
```

- A função `enumFrom` produz uma lista sem especificar um limite superior, logo o resultado pode ser uma lista infinita.

```
Prelude*> take 5 [0..]
[0,1,2,3,4]
```

Intervalos

```
enumFromThenTo :: Enum a => a -> a -> a -> [a]
```

- A função `enumFromThenTo` produz uma lista em que cada elemento adjacente possui a mesma "distância".

```
[1.0, 1.2 .. 2.0]
[1.0,1.2,1.4,1.5999999999999999,1.7999999999999998,1.99999
```

Derivação automática

- Instâncias de Enum podem ser calculadas automaticamente para enumerações.

```
data Direction
   = North | South | East | West
     deriving (Eq, Ord, Show, Enum)
```

Exercícios

- Implemente uma instância de Eq para o tipo `Vec3`.
- Implemente uma instância de Eq para o tipo `Person` que considere duas pessoas iguais se essas possuírem o mesmo nome.

```
data Person
  = Person {
      name :: String
    , age  :: Int
    }
```

Exercícios

- Implemente uma instância de Show para `Person` de forma que o resultado da conversão para string de um valor de tipo `Person` exiba apenas o campo name deste registro.