Polimorfismo *ad hoc*
Type Classes

Vander Alves

Classes

- Usadas para permitir overloading de nomes
  - operação de igualdade ==
    - diferentes significados para diferentes tipos
- Orientação a Objetos
  - herança

Funções polimórficas?

```
(==) :: t -> t -> Bool

(<) :: t -> t -> Bool

show :: t -> String
```

Funções monomórficas

- Definição funciona apenas para um tipo de dados específico

```
capitalize :: Char -> Char
capitalize ch = chr (ord ch + offset)
  where offset = ord 'A' - ord 'a'
```

Funções polimórficas

- Uma única definição pode ser usada para diversos tipos de dados

```
length :: [a] ->Int
length [] = 0
length (x:xs) = 1 + length xs
```

Overloading

- A função pode ser usada para vários (alguns) tipos de dados
  - diferentes definições para cada tipo
- Classe: coleção de tipos para os quais uma função está definida
- O conjunto de tipos para os quais (==) está definida é a *classe igualdade*, `Eq`

Definindo a classe igualdade

- Identifica-se o que é necessário para um tipo `t` ser da classe
  - nesse caso, ele deve possuir uma função (==) definida sobre `t`, do tipo `t -> t -> Bool`

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

Instâncias

- Tipos membros de uma classe são chamadas instâncias
- São instâncias de `Eq` os tipos primitivos e as listas e tuplas de instâncias de `Eq`
  - `Int`, `Float`, `Char`, `Bool`, `[Int]`, `(Int,Bool)`,`[[Char]]`, `[(Int,[Bool])]`
- `Int -> Int` não é instância da classe `Eq`
- Instância de classe vs. Instância de um tipo

Funções que usam igualdade

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

- Contexto
  - definido pela parte antes do operador "=>"

```
allEqual succ succ succ     ?
```

```
member :: Eq t => [t] -> t -> Bool
member [] b = False
member (a:as) b = (a==b) || member as b
```

Outras funções

```
books :: Eq a => [(a,b)] -> a -> [b]

borrowed :: Eq u => [(t,u)] -> u -> Bool

numBorrowed :: Eq t => [(t,u)] -> t -> Int
```

Definido assinaturas de classes

- Funções (nome e tipo) que devem ser definidas para cada instância da classe
- Exemplo de uma assinatura de classe

```
class Visible t where
   toString :: t -> String
   size :: t -> Int
```

Definindo instâncias de uma classe

- Definir as funções da assinatura para um tipo
- Exemplo de uma instância da classe `Eq`

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

Exemplo de instâncias da classe `Visible`

```
instance Visible Char where
  toString ch = [ch]
  size _ = 1

instance Visible Bool where
  toString True  = "True"
  toString False = "False"
  size _ = 1
```

Exemplo de instâncias da classe `Visible`

```
instance Visible t => Visible [t] where
  toString = map toString >.> concat
  size = map size >.> foldr (+) 0
```

Definições default

```
class Eq t where
    (==), (/=) :: t -> t -> Bool
    a /= b = not (a==b)
```

podem ser substituídas (*overloaded*)

Classes derivadas

```
class Eq t => Ord t where
   (<),(<=),(>),(>=) :: t -> t -> Bool
   max, min :: t -> t -> t
   a <= b = (a < b || a == b)
   a > b  = b < a
   a < b  = b > a
   a >= b = (a > b || a == b)
iSort :: Ord t => [t] ->[t]
```

Herança de operações

Restrições múltiplas

```
vSort = iSort >.> toString
vSort :: (Ord t, Visible t) => [t] -> String

instance (Eq t, Eq u) => Eq (t,u) where
  (a,b) == (c,d) = (a == c && b == d)

class (Ord t, Visible t) => OrdVis t
```

Herança múltipla

Classes predefinidas

```
Eq, Ord

class (Ord t) => Enum t where
  enumFrom :: t -> [t]
  enumFromThen :: t -> t -> [t]
  enumFromTo :: t -> t -> [t]
  enumFromThenTo :: t -> t -> t -> [t]

[n .. ]
[n,m .. ]
[n .. m]
[n,n' .. m]
```

Classes predefinidas

```
show :: (Show t) => t -> String
read :: (Read t) => String -> t
```

Tipos Numéricos

```
Int, Integer
Float, Double
Rational
Complex
```

Classes `Num, Fractional`

```
rep 0 ch = []
rep n ch = ch : rep (n-1) ch
:type rep
rep :: Num a => a -> b -> [b]
```