Classe de Tipos

Wladimir Araújo Tavares [1]

[1]Universidade Federal do Ceará - Campus de Quixadá

2 de junho de 2016

Classe Num

- A definição mínima completa da classe Num é formada pelas seguintes funções: (+), (*), abs, signum, fromInteger, (negate | (-))
- `negate :: a -> a`: negação únaria
- `abs :: a -> a`: valor absoluto
- `signum :: a -> a`: sinal do número. Para números reais, signum devolve -1 para números negativos, 0 para zero e 1 para números positivos. A função signum e abs satisfazem a seguinte propriedade:

```
signum x * abs x == x
```

- `fromInteger :: Integer -> a`: conversão a partir de um inteiro.

Class Num

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

Classe Show

- A definição mínima completa da classe Show é formada pela função: show
- `show :: a -> String`: devolve uma String simples.

Classe Eq

- A classe Eq define a igualdade (==) e desigualdade (/=). Todos os tipos básicos exportados pelo Prelúdio são instâncias de Eq. Eq pode ser derivado para qualquer tipo definido cujos elementos são instâncias da classe Eq.
- A definição mínima completa da classe Show é formada pela função: (==) | (=/)
- `(==) :: a -> a -> Bool`: verifica se os tipos são iguais.

Classe Ord

- A classe Ord é usada para tipos de dados totalmente ordenados. Uma instância de Ord pode ser derivada para qualquer tipo definido cujos elementos estão em Ord.
- A definição mínima completa da classe Show é formada pela função: compare | (<=)
- `(<=) :: a -> a -> Bool`: verifica se o primeiro é menor ou igual ao segundo.

Tipo Inteiro

Torne o tipo Inteiro uma instância da classe Num. As funções sobrecarregadas em num devem ter os seguintes comportamentos
`data Inteiro = Zero | Succ Inteiro | Pred Inteiro`

- (+): some dois inteiro Inteiro.
- (*): multiplique dois inteiros Inteiro.
- negate: multiplique por -1.
- abs, signum: devem satisfazer a seguinte propriedade : `abs x * signum x == x`
- fromInteger z: retorne a inteiro Inteiro.

Tipo Inteiro

```
data Inteiro = Zero | Succ Inteiro | Pred Inteiro

toInt :: Inteiro -> Integer
toInt Zero     = 0
toInt (Succ x) = toInt x + 1
toInt (Pred x) = toInt x - 1

fromInt 0 = Zero
fromInt x | x > 0     = Succ (fromInt (x-1))
  | otherwise = Pred (fromInt (x+1))
```

```
instance Show Inteiro where
show x = show (toInt x)

instance Num Z where
z1 + z2       = fromInt ( (toInt z1) + (toInt z2) )
z1 * z2       = fromInt ( (toInt z1) * (toInt z2) )
negate z      = fromInt ( negate (toInt z) )
abs z         = fromInt ( abs (toInt z) )
signum z      = fromInt ( signum (toInt z) )
fromInteger z = fromInt z
```

```
instance Show Inteiro where
show x = show (toInt x)

instance  Eq Inteiro where
z1 == z2 = (toInt z1) == (toInt z2)

instance Ord Inteiro where
z1 <= z2 = (toInt z1) <= (toInt z2)
```

Fraction

Torne Fraction uma instância da classe Num. As funções sobrecarregadas em num devem ter os seguintes comportamentos

```
data Fraction = Q Integer Integer
```

- $(+)$: some duas frações.
- $(*)$: multiplique duas frações.
- negate: inverter os sinais do numerador da fração.
- abs: retorne a fração $\frac{|x|}{y}$.
- signum: retorne a fração $\frac{sigum(x)}{1}$.
- fromInteger z: retorne a fração $\frac{z}{1}$

```
module Fraction(Fraction) where
import GHC.Real
data Fraction = Q Integer Integer
instance  Show Fraction where
show (Q x y) = show x ++ "/" ++ show y ++ "(" ++ show ( (
    fromIntegral x) / (fromIntegral y) ) ++ ")"

instance Num Fraction where
(Q x y) + (Q z w) = Q (x*w + y*z) (y*w)
(Q x y) * (Q z w) = Q (x*z) (y*w)
negate (Q x y)  = Q (negate x) y
abs    (Q x y)  = Q (abs x) y
signum (Q x y)  = Q (signum x) 1
fromInteger z   = Q z 1
```

```
instance Fractional Fraction where
f1 / f2 = f1 * (recip f2)
  recip (Q a b)  = Q b a
fromRational r = Q (numerator r) (denominator r)

instance Eq Fraction where
(Q a b) == (Q c d) = a*d == b*c

instance Ord Fraction where
(Q a b) <= (Q c d) = a*d <= b*c
```