

Herança

O sistema de classes do Haskell tem um mecanismo de **herança**. Por exemplo, podemos definir a classe **Ord** como uma **extensão** da classe **Eq**.

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

- A classe **Ord** **herda** todas as funções da classe **Eq** e, além disso, estabelece um conjunto de operações de comparação e as funções máximo e mínimo.
- Diz-se que **Eq** é uma **superclasse** de **Ord**, ou que **Ord** é uma **subclasse** de **Eq**.
- Todo o tipo que é instância de **Ord** tem necessariamente de ser instância de **Eq**.

161

A classe Ord

```
data Ordering = LT | EQ | GT
```

```
class (Eq a) => Ord a where
  compare :: a -> a -> Ordering
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min :: a -> a -> a
  -- Minimal complete definition: (<=) or compare
  -- using compare can be more efficient for complex types
  compare x y | x == y = EQ
               | x <= y = LT
               | otherwise = GT
  x <= y = compare x y /= GT
  x < y = compare x y == LT
  x >= y = compare x y /= LT
  x > y = compare x y == GT
  max x y | x <= y = y
           | otherwise = x
  min x y | x <= y = x
           | otherwise = y
```

Para declarar um tipo como instância da classe **Ord**, basta definir a função **(<=)** ou a função **compare**

162

A classe Ord

Exemplo: Declarar **Nat** como instância da classe **Ord**

```
data Nat = Zero | Suc Nat
  deriving (Eq)
```

pode ser feito assim:

```
instance Ord Nat where
  compare (Suc _) Zero = GT
  compare Zero (Suc _) = LT
  compare Zero Zero = EQ
  compare (Suc x) (Suc y) = compare x y
```

```
> Suc Zero <= Suc (Suc Zero)
True
```

A função **(<=)** fica definida por omissão.

Ou, **em alternativa**, assim:

```
instance Ord Nat where
  Zero <= _ = True
  (Suc x) <= (Suc y) = x <= y
  (Suc _) <= Zero = False
```

163

A classe Ord

Exemplo: Declarar **Time** como instância da classe **Ord**

```
data Time = AM Int Int
           | PM Int Int
           | Total Int Int
```

```
totalmin :: Time -> Int
totalmin (AM h m) = h*60 + m
totalmin (PM h m) = (12+h)*60 + m
totalmin (Total h m) = h*60 + m
```

```
instance Eq Time where
  t1 == t2 = (totalmin t1) == (totalmin t2)
```

É necessário que **Time** seja da classe **Eq**.

pode agora ser feito assim:

```
instance Ord Time where
  t1 <= t2 = (totalmin t1) <= (totalmin t2)
```

Exemplo de uma função que usa o operador **(<)** definido por omissão:

```
select :: Time -> [(Time,String)] -> [(Time,String)]
select t l = filter ((t<) . fst) l
```

Este é o **(<=)** para o tipo **Int**. Note que **Int** é instância da classe **Ord**.

164

A classe Show

A classe **Show** estabelece métodos para converter um valor de um tipo qualquer numa string.

O interpretador Haskell usa a função **show** para apresentar o resultado dos seu cálculos.

```
class Show a where
  show      :: a -> String
  showsPrec :: Int -> a -> ShowS
  showList  :: [a] -> ShowS
  -- Minimal complete definition: show or showsPrec
  show x      = showsPrec 0 x ""
  showsPrec _ x s = show x ++ s
  showList [] = showString "[]"
  showList (x:xs) = showChar '[' . shows x . showList xs
    where showList [] = showChar ']'
          showList (x:xs) = showChar ',' . shows x . showList xs
```

Basta definir a função **show**. O restante fica definido por omissão.

```
type ShowS = String -> String
```

```
shows :: Show a => a -> ShowS
shows = showsPrec 0
```

A função **showsPrec** usa uma string como acumulador. É mais eficiente.

165

A classe Show

Exemplo: Declarar **Nat** como instância da classe **Show** de forma a que os naturais sejam apresentados do modo usual

```
natToInt :: Nat -> Int
natToInt Zero = 0
natToInt (Suc n) = 1 + (natToInt n)
```

```
> Suc (Suc Zero)
2
```

```
instance Show Nat where
  show n = show (natToInt n)
```

Este é o **show** para o tipo **Int**. Note que **Int** é instância da classe **Show**.

Instâncias da classe **Show** podem ser derivadas automaticamente. Neste caso, o método **show** produz uma string com o mesmo aspecto do valor que lhe é passado como argumento.

Ficaria assim:

```
data Nat = Zero | Suc Nat
deriving (Eq, Show)
```

```
> Suc (Suc Zero)
Suc (Suc Zero)
```

166

A classe Show

Exemplo: Declarar **Time** como instância da classe **Show**

```
instance Show Time where
  show (AM h m) = (show h) ++ ":" ++ (show m) ++ " am"
  show (PM h m) = (show h) ++ ":" ++ (show m) ++ " pm"
  show (Total h m) = (show h) ++ "h" ++ (show m) ++ "m"
```

```
> AM 4 30
4:30 am
> Total 17 45
17h45m
```

A função **show** é usada para apresentar o valor no ghci.

A função **showList**, definida por omissão, é usada pelo ghci para apresentar a lista.

```
> [(PM 43 20), (AM 2 15), (Total 17 30)]
[43:20 pm, 2:15 am, 17h30m]
```

167

A classe Num

A classe **Num** está no topo de uma hierarquia de classes numéricas desenhada para controlar as operações que devem estar definidas sobre os diferentes tipos de números. Os tipos **Int**, **Integer**, **Float** e **Double**, são instâncias desta classe.

Note que **Num** é subclasse das classes **Eq** e **Show**.

```
class (Eq a, Show a) => Num a where
  (+), (-), (*) :: a -> a -> a
  negate      :: a -> a
  abs, signum :: a -> a
  fromInteger :: Integer -> a
  -- Minimal complete definition: All, except negate or (-)
  x - y      = x + negate y
  negate x   = 0 - x
```

A função **fromInteger** converte um **Integer** num valor do tipo **Num a => a**.

```
> :type 35
35 :: Num a => a
> 35 + 5.7
40.7
```

35 é na realidade (**fromInteger 35**)

168

A classe Num

Exemplo: Nat como instância da classe Num

Note que **Nat** já é das classes **Eq** e **Show**.

```
somaNat :: Nat -> Nat -> Nat
somaNat Zero n    = n
somaNat (Suc n) m = Suc (somaNat n m)
```

```
prodNat :: Nat -> Nat -> Nat
prodNat Zero _    = Zero
prodNat (Suc n) m = somaNat m (prodNat n m)
```

```
subtNat :: Nat -> Nat -> Nat
subtNat n Zero    = n
subtNat (Suc n) (Suc m) = subtNat n m
subtNat Zero _    = error "indefinido ..."
```

```
instance Num Nat where
  (+) = somaNat
  (*) = prodNat
  (-) = subtNat
  fromInteger = deInteger
  abs = id
  signum = sinal
  negate n = error "indefinido ..."
```

```
deInteger :: Integer -> Nat
deInteger 0    = Zero
deInteger n | n>0 = Suc (deInteger (n-1))
             | n<0 = error "indefinido ..."
```

```
sinal :: Nat -> Nat
sinal Zero    = Zero
sinal (Suc _) = Suc Zero
```

```
> dois = Suc (Suc Zero)
> dois * dois
4
```

169

A classe Enum

A classe **Enum** estabelece um conjunto de operações que permitem **seqüências aritméticas**.

```
class Enum a where
  succ, pred      :: a -> a
  toEnum          :: Int -> a
  fromEnum        :: a -> Int
  enumFrom        :: a -> [a]           -- [n..]
  enumFromThen    :: a -> a -> [a]      -- [n,m..]
  enumFromTo      :: a -> a -> [a]      -- [n..m]
  enumFromThenTo  :: a -> a -> a -> [a] -- [n,n'..m]
  -- Minimal complete definition: toEnum, fromEnum
  succ            = toEnum . (1+)
  pred            = toEnum . subtract 1
  enumFrom x      = map toEnum [ fromEnum x .. ]
  enumFromThen x y = map toEnum [ fromEnum x, fromEnum y .. ]
  enumFromTo x y  = map toEnum [ fromEnum x .. fromEnum y ]
  enumFromThenTo x y z = map toEnum [ fromEnum x, fromEnum y .. fromEnum z ]
```

Entre as instâncias desta classe contam-se os tipos: **Int**, **Integer**, **Float**, **Double**, **Char**, ...

```
> [2,2.5 .. 4]
[2.0,2.5,3.0,3.5,4.0]
```

```
> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
```

170

A classe Enum

Exemplo: Time como instância da classe Enum

```
instance Enum Time where
  toEnum n = let (h,m) = divMod n 60
              in Total h m
  fromEnum = totalmin
```

```
> [(AM 1 0), (AM 2 30) .. (PM 1 0)]
[1h0m,2h30m,4h0m,5h30m,7h0m,8h30m,10h0m,11h30m,13h0m]
> [(PM 2 25) .. (Total 14 30)]
[14h25m,14h26m,14h27m,14h28m,14h29m,14h30m]
```

É possível derivar automaticamente instâncias da classe **Enum**, apenas em **tipos enumerados**.

Exemplo:

```
data Cor = Amarelo | Verde | Vermelho | Azul
         deriving (Enum, Show)
```

```
> [Amarelo .. Azul]
[Amarelo,Verde,Vermelho,Azul]
```

171

A classe Read

A classe **Read** estabelece funções que são usadas na conversão de uma string num valor do tipo de dados que é instância de **Read** (quando isso é possível).

```
class Read a where
  readsPrec :: Int -> ReadS a
  readList  :: ReadS [a]
  -- Minimal complete definition: readsPrec
  readList  = ...
```

```
read :: Read a => String -> a
read s = case [x | (x,t) <- reads s, ("","") <- lex t] of
  [x] -> x
  []  -> error "Prelude.read: no parse"
  _   -> error "Prelude.read: ambiguous parse"
```

```
type ReadS a = String -> [(a,String)]
```

```
reads :: Read a => ReadS a
reads = readsPrec 0
```

lex é um analisador léxico do Prelude.

Podemos definir instâncias da classe **Read** que permitam fazer o **parser** do texto de acordo com uma determinada sintaxe, mas isso **não é** tópico de estudo nesta disciplina.

172