

Polimorfismo *ad hoc* (sobrecarga)

O Haskell incorpora ainda uma outra forma de polimorfismo que é a **sobrecarga de funções**. Um mesmo identificador de função pode ser usado para designar funções computacionalmente distintas. A esta característica também se chama **polimorfismo *ad hoc***.

Exemplos:

O operador (+) tem sido usado para somar, tanto valores inteiros como valores decimais.

O operador (==) pode ser usado para comparar inteiros, caracteres, listas de inteiros, strings, booleanos, ...

Afinal, qual é o tipo de (+) ? E de (==) ?

A sugestão `(+) :: a -> a -> a` **não serve**, pois são tipos demasiado genéricos !
`(==) :: a -> a -> Bool`

Faria com que fossem aceites expressões como, por exemplo:

`('a' + 'b')` , `(True + False)` , `("esta" + "errado")` ou `(div == mod)` ,

e estas expressões resultariam em **erro**, pois estas operações não estão preparadas para trabalhar com valores destes tipos.

Em Haskell esta situação é resolvida através de **tipos qualificados** (*qualified types*), fazendo uso da noção de **classe**.

113

Tipos qualificados

Conceptualmente, um **tipo qualificado** pode ser visto como um tipo polimórfico só que, em vez da quantificação universal da forma "para todo o tipo a, ..." vai-se poder dizer "para todo o tipo a que pertence à classe C, ..." . Uma classe pode ser vista como um conjunto de tipos.

Exemplo:

Sendo **Num** uma classe (*a classe dos números*) que tem como elementos os tipos: Int, Integer, Float, Double, ..., pode-se dar a (+) o tipo preciso de:

$$\forall a \in \text{Num}. a \rightarrow a \rightarrow a$$

o que em Haskell se vai escrever: `(+) :: Num a => a -> a -> a`

e lê-se: "para todo o tipo a que pertence à classe Num, (+) tem tipo a->a->a".

Uma classe surge assim como uma forma de classificar tipos (quanto às funcionalidades que lhe estão associadas). Neste sentido as classes podem ser vistas como os **tipos dos tipos**.

Os tipos que pertencem a uma classe também serão chamados de **instâncias** da classe.

A capacidade de **qualificar** tipos polimórficos é uma característica inovadora do Haskell.

114

Classes & Instâncias

Uma **classe** estabelece um conjunto de assinaturas de funções (os **métodos da classe**). Os tipos que são declarados como **instâncias** dessa classe têm que ter definidas essas funções.

Exemplo: A seguinte declaração (simplificada) da classe **Num**

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

impõe que todo o tipo **a** da classe **Num** tenha que ter as operações (+) e (*) definidas.

Para declarar **Int** e **Float** como elementos da classe **Num**, tem que se fazer as seguintes **declarações de instância**

```
instance Num Int where
  (+) = primPlusInt
  (*) = primMulInt
```

```
instance Num Float where
  (+) = primPlusFloat
  (*) = primMulFloat
```

Neste caso as funções *primPlusInt*, *primMulInt*, *primPlusFloat* e *primMulFloat* são funções primitivas da linguagem.

Se `x::Int` e `y::Int` então `x + y` \equiv `x `primPlusInt` y`
Se `x::Float` e `y::Float` então `x + y` \equiv `x `primPlusFloat` y`

115

Tipo principal

O **tipo principal** de uma expressão ou de uma função é o tipo mais geral que lhe é possível associar, de forma a que todas as possíveis instâncias desse tipo constituam ainda tipos válidos para a expressão ou função.

Qualquer expressão ou função válida tem um tipo principal **único**. O Haskell **infer** sempre o tipo principal das expressões ou funções, mas é sempre possível associar tipos mais específicos (que são instância do tipo principal).

Exemplo: O tipo principal inferido pelo Haskell para o operador (+) é

```
(+) :: Num a => a -> a -> a
```

Mas, `(+) :: Int -> Int -> Int` e `(+) :: Float -> Float -> Float` são também tipos válidos dado que tanto Int como Float são instâncias da classe Num, e portanto podem substituir a variável a.

Note que **Num a** não é um tipo, mas antes uma restrição sobre um tipo. Diz-se que (**Num a**) é o **contexto** para o tipo apresentado.

Exemplo:

```
sum [] = 0
sum (x:xs) = x + sum xs
```

O tipo principal da função sum é

```
sum :: Num a => [a] -> a
```

- `sum :: [a] -> a` seria um tipo demasiado geral. **Porquê ?**
- **Qual será o tipo principal da função product ?**

116

Definições por defeito

Relembre a definição da função pré-definida elem:

```
elem x [] = False
elem x (y:ys) = (x==y) || elem x ys
```

Qual será o seu tipo ?

É necessário que (==) esteja definido para o tipo dos elementos da lista.

Existe pré-definida a classe **Eq**, dos tipos para os quais existe uma operação de igualdade.

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  -- Minimal complete definition: (==) or (/=)
  x == y = not (x /= y)
  x /= y = not (x == y)
```

Esta classe estabelece as funções (==) e (/=) e, para além disso, fornece também **definições por defeito** para estes métodos (*default methods*).

Caso a definição de uma função seja omitida numa declaração de instância, o sistema assume a definição por defeito feita na classe. Se existir uma nova definição do método na declaração de instância, será essa definição a ser usada.

117

Instâncias com restrições

Relembre a definição das árvores binárias.

```
data ArvBin a = Vazia
              | Nodo a (ArvBin a) (ArvBin a)
```

Como poderemos fazer o teste de igualdade para árvores binárias ?

Duas árvores são iguais se tiverem a mesma estrutura (a mesma forma) e se os valores que estão nos nodos também forem iguais.

Portanto, para fazer o teste de igualdade em (**ArvBin a**), necessariamente, tem que se saber como testar a igualdade entre os valores que estão nos nodos, i.e., em **a**.

Só poderemos declarar (**ArvBin a**) como instância da classe **Eq** se **a** for também uma instância da classe **Eq**.

Este tipo de *restrição* pode ser colocado na declaração de instância, fazendo:

```
instance (Eq a) => Eq (ArvBin a) where
  Vazia == Vazia = True
  (Nodo x1 e1 d1) == (Nodo x2 e2 d2) = (x1==x2) && (e1==e2)
                                     && (d1==d2)
  _ == _ = False
```

(==) de **a** (==) de (**ArvBin a**)

119

Exemplos de instâncias de Eq

O tipo **Cor** é uma instância da classe Eq com (==) definido como se segue:

```
instance Eq Cor where
  Azul == Azul = True
  Verde == Verde = True
  Amarelo == Amarelo = True
  Vermelho == Vermelho = True
  _ == _ = False
```

O método (/=) está definido por defeito.

O tipo **Nat** também pode ser declarado como instância da classe Eq:

```
instance Eq Nat where
  (Suc n) == (Suc m) = n == m
  Zero == Zero = True
  _ == _ = False
```

(==) de **Nat**

O tipo **PontoC** com instância de Eq:

```
instance Eq PontoC where
  (Pt x1 y1 c1) == (Pt x2 y2 c2) = (x1==x2) && (y1==y2)
                                     && (c1==c2)
```

(==) de **Float**

(==) de **Cor**

Nota: (==) é uma função recursiva em **Nat**, mas não em **PontoC**.

118

Instâncias derivadas de Eq

O testes de igualdade definidos até aqui implementam a **igualdade estrutural** (dois valores são iguais quando resultam do mesmo construtor aplicado a argumentos também iguais).

Quando assim é pode-se evitar a declaração de instância se na declaração do tipo for acrescentada a instrução **deriving Eq**.

Exemplos: Com estas declarações, o Haskell deriva automaticamente declarações de instância de **Eq** (iguais às que foram feitas) para estes tipos.

```
data Cor = Azul | Amarelo | Verde | Vermelho
  deriving Eq
```

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

```
data PontoC = Pt {xx :: Float, yy :: Float, cor :: Cor}
  deriving Eq
```

```
data ArvBin a = Vazia
              | Nodo a (ArvBin a) (ArvBin a)
  deriving Eq
```

120

Mas, nem sempre a igualdade estrutural é a desejada.

Exemplo: Relembre o tipo de dados `Figura`:

```
data Figura = Rectangulo Float Float
            | Circulo Float
            | Triangulo Float Float
```



Neste caso queremos que duas figuras sejam consideradas iguais ainda que a ordem pela qual os valores são passados possa ser diferente.

```
instance Eq Figura where
  (Rectangulo x1 y1) == (Rectangulo x2 y2) =
    ((x1==x2) && (y1==y2)) || ((x1==y2) && (x2==y1))
  (Circulo r1) == (Circulo r2) = r1==r2
  (Triangulo x1 y1) == (Triangulo x2 y2) =
    ((x1==x2) && (y1==y2)) || ((x1==y2) && (x2==y1))
```

121

Herança

O sistema de classes do Haskell também suporta a noção de **herança**.

Exemplo: Podemos definir a classe `Ord` como uma **extensão** da classe `Eq`.

-- isto é uma simplificação da classe Ord já pré-definida

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

A classe `Ord` herda todos os métodos de `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 que ser instância de `Eq`.

Exemplo:

```
estaABProc :: Ord a => a -> ArvBin a -> Bool
estaABProc _ Vazia = False
estaABProc x (Nodo y e d) | x < y = estaABProc x e
                          | x > y = estaABProc x d
                          | x == y = True
```

A restrição `(Eq a)` não é necessária. Porquê?

123

Exercícios:

- Considere a seguinte definição de tipo, para representar horas nos dois formatos usuais.

```
data Time = Am Int Int
          | Pm Int Int
          | Total Int Int
```

Declare `Time` como instância da classe `Eq` de forma a que `(==)` teste se dois valores representam a mesma hora do dia, independentemente do seu formato.

- Qual o tipo principal da seguinte função:

```
lookup x ((y,z):yzs) | x /= y = lookup x yzs
                  | otherwise = Just z
lookup _ [] = Nothing
```

- Considere a seguinte declaração: `type Assoc a b = [(a,b)]`

Será que podemos declarar `(Assoc a b)` como instância da classe `Eq`?

122

Herança múltipla

O sistema de classes do Haskell também suporta **herança múltipla**. Isto é, uma classe pode ter mais do que uma superclasse.

Exemplo: A classe `Real`, já pré-definida, tem a seguinte declaração

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

A classe `Real` herda todos os métodos da classe `Num` e da classe `Ord` e estabelece mais uma função.

NOTA: Na declaração dos tipos dos métodos de uma classe, é possível colocar restrições às variáveis de tipo, excepto à variável de tipo da classe que está a ser definida.

Exemplo:

```
class C a where
  m1 :: Eq b => (b,b) -> a -> a
  m2 :: Ord b => a -> b -> b -> a
```

O método `m1` impõe que `b` pertença à classe `Eq`, e o método `m2` impõe que `b` pertença a `Ord`. Restrições à variável `a`, se forem necessárias, terão que ser feitas no contexto da classe, e nunca ao nível dos métodos.

124

A classe Ord

```
data Ordering = LT | EQ | GT
  deriving (Eq, Ord, Ix, Enum, Read, Show, Bounded)
```

```
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
```

125

As restrições às variáveis de tipo que são impostas pelo contexto, *propagam-se* ao logo do processo de inferência de tipos do Haskell.

Exemplo: Relembre a definição da função quicksort.

```
parte :: (Ord a) => a -> [a] -> ([a],[a])
parte _ [] = ([],[a])
parte x (y:ys) | y < x = (y:as,bs)
                | otherwise = (as,y:bs)
  where (as,bs) = parte x ys
```

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) = let (l1,l2) = parte x xs
                    in (quicksort l1)++[x]++(quicksort l2)
```

Note como o contexto **(Ord a)** do tipo da função **parte** se propaga para a função **quicksort**.

127

Exemplos de instâncias de Ord

Exemplo:

```
instance Ord Nat where
  compare (Suc _) Zero = GT
  compare Zero (Suc _) = LT
  compare Zero Zero    = EQ
  compare (Suc n) (Suc m) = compare n m
```

Instâncias da classe **Ord** podem ser *derivadas automaticamente*. Neste caso, a relação de ordem é estabelecida com base na ordem em que os construtores são apresentados e na relação de ordem entre os parâmetros dos construtores.

Exemplo:

```
data AB a = V | NO a (AB a) (AB a)
  deriving (Eq,Ord)
```

```
ar1 = NO 1 V V
ar2 = NO 2 V V
```

Será que poderíamos não derivar Eq ?

```
> V < ar1
True
> ar1 < ar2
True
> (NO 4 ar1 ar2) < (NO 5 ar2 ar1)
True
> (NO 4 ar1 ar2) < (NO 3 ar2 ar1)
False
> (NO 4 ar1 ar2) < (NO 4 ar2 ar1)
True
```

126

A classe Show

A classe **Show** estabelece métodos para converter um valor de um tipo qualquer (que lhe pertença) numa string.

O interpretador Haskell usa o método **show** para apresentar o resultado dos seus 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
```

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

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

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

128

Exemplos de instâncias de Show

Exemplo:

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

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

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

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.

Exemplo: Se, em alternativa, tivéssemos feito

```
data Nat = Zero | Suc Nat
  deriving Show
```

teríamos

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

Exemplo:

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

```
> (AM 9 30)
9:30 am
```

```
> (PM 1 35)
1:35 pm
```

129

Exemplos de instâncias de Num

Exemplo:

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

Note que **Nat** já pertence às classes **Eq** e **Show**.

```
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 ..."
```

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

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

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

131

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.

```
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**.

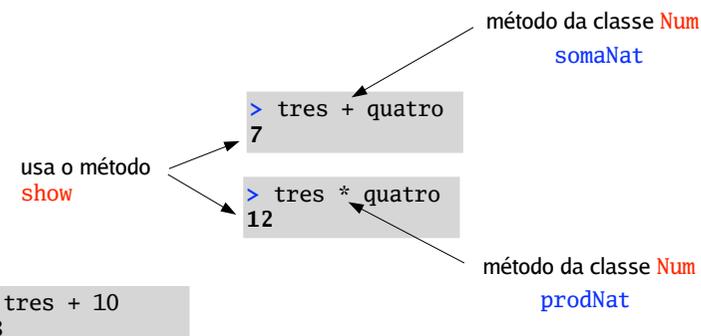
```
Prelude> :t 35
35 :: Num a => a
```

35 é na realidade (fromInteger 35)

```
Prelude> 35 + 2.1
37.1
```

130

```
tres = Suc (Suc (Suc Zero))
quatro = Suc tres
```



Nota: Não é possível derivar automaticamente instâncias da classe **Num**.

132

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`, `Char`, `Bool`, ...

Exemplos: `Prelude> [2,2.5 .. 4]` `Prelude> ['a'..'z']`
`[2.0,2.5,3.0,3.5,4.0]` `"abcdefghijklmnopqrstuvwxy"`

133

A classe Read

A classe `Read` estabelece funções que são usadas na conversão de uma string num valor do tipo de dados (instância de `Read`).

```
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* definido no `Prelude`.

135

Exemplos de instâncias de Enum

Exemplo:

```
instance Enum Nat where
  toEnum = intToNat
  where intToNat :: Int -> Nat
        intToNat 0 = Zero
        intToNat (n+1) = Suc (intToNat n)

  fromEnum = natToInt
```

```
> [Zero, tres .. (tres * tres)]
[0,3,6,9]
> [Zero .. tres]
[0,1,2,3]
> [(Suc Zero), tres ..]
[1,3,6,9,12,15,18,21,23,25, ...]
```

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

Exemplo:

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

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

134

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.*)

Instâncias da classe `Read` podem ser *derivadas automaticamente*. Neste caso, a função `read` recebendo uma string que obedeça às regras sintáticas de Haskell produz o valor do tipo correspondente.

Exemplos:

```
data Time = Am Int Int
          | Pm Int Int
          | Total Int Int
  deriving (Show, Read)
```

```
data Nat = Zero | Suc Nat
  deriving Read
```

```
> read "Am 8 30" :: Time
Am 8 30
> read "(Total 17 15)" :: Time
Total 17 15
> read "Suc (Suc Zero)" :: Nat
2
> read "[2,3,6,7]" :: [Int]
[2,3,6,7]
> read "[Zero, Suc Zero]" :: [Nat]
[0,1]
```

É necessário indicar o tipo do valor a produzir.

Quase todos os tipos pré-definidos pertencem à classe `Read`.

Porquê ?

136

Declaração de tipos polimórficos com restrições nos parâmetros

Na declaração de um **tipo algébrico** pode-se exigir que os parâmetros pertençam a determinadas classes.

Exemplo:

```
data (Ord a) => STree a = Null
    | Branch a (STree a) (STree a)
```

```
delSTree x Null = Null
delSTree x (Branch y e Null) | x == y = e
delSTree x (Branch y Null d) | x == y = d
delSTree x (Branch y e d)
    | x < y = Branch y (delSTree x e) d
    | x > y = Branch y e (delSTree x d)
    | x == y = let z = minSTree d
                in Branch z e (delSTree z d)
```

```
minSTree (Branch x Null _) = x
minSTree (Branch _ e _) = minSTree e
```

Na declaração de **tipos sinónimos** também se podem impôr restrições de classes.

Exemplo:

```
type TAssoc a b = (Eq a) => [(a,b)]
```

137

Classes de Construtores de Tipos

Relembre os tipos paramétricos (`Maybe a`), `[a]`, `(ArvBin a)`, `(Tree a)` ou `(ABin a b)`. `Maybe`, `[]`, `ArvBin`, `Tree` e `ABin`, não são tipos, mas podem ser vistos como operadores sobre tipos – são **construtores de tipos**.

Exemplo: `Maybe` não é um tipo, mas `(Maybe Int)` é um tipo que resulta de aplicar o construtor de tipos `Maybe` ao tipo `Int`.

Em Haskell é possível definir classes de construtores de tipos. Um exemplo disso é a classe **Functor**:

```
class Functor f where
    fmap :: (a -> b) -> (f a -> f b)
```

Note que `f` não é um tipo.
`f a` e `f b` é que são tipos.

Exemplos:

```
instance Functor [] where
    fmap = map
```

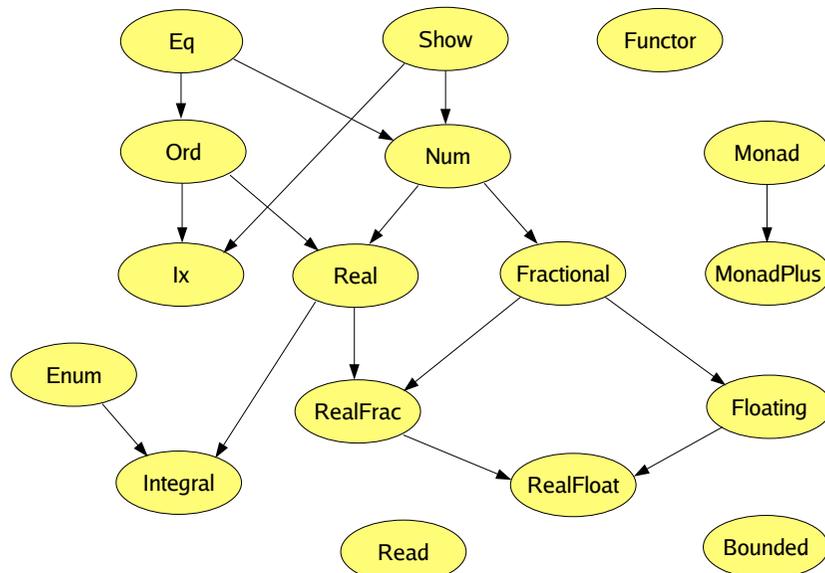
```
instance Functor Maybe where
    fmap f Nothing = Nothing
    fmap f (Just x) = Just (f x)
```

Note que o que se está a declarar como instância da classe **Functor** são construtores de tipos.

```
instance Functor ArvBin where
    fmap = mapAB
```

139

Hierarquia de classes pré-definidas do Haskell



Prelude> `:i Nome_da_Classe`

138

Definição de novas classes

Para além da hierarquia de classes pré-definidas, o Haskell permite **definir novas classes**.

Exemplo: Podemos definir a classe das *ordens parciais* da seguinte forma

```
class (Eq a) => OrdParcial a where
    comp :: a -> a -> Maybe Ordering -- basta definir comp

    lt, gt, eq :: a -> a -> Maybe Bool
    lt x y = case (comp x y)
    of { Nothing -> Nothing ; (Just LT) -> Just True ; _ -> Just False }
    gt x y = case (comp x y)
    of { Nothing -> Nothing ; (Just GT) -> Just True ; _ -> Just False }
    eq x y = case (comp x y)
    of { Nothing -> Nothing ; (Just EQ) -> Just True ; _ -> Just False }

    maxi, mini :: a -> a -> Maybe a
    maxi x y = case (comp x y) of
        Nothing -> Nothing
        Just GT -> Just x
        _ -> Just y
    mini x y = case (comp x y) of
        Nothing -> Nothing
        Just LT -> Just x
        _ -> Just y
```

Nota: Repare nos diversos modos de escrever expressões case.

140

A relação de *inclusão de conjuntos* é um bom exemplo de uma relação de ordem parcial.

Exemplo: A noção de conjunto pode ser implementada pelo tipo

```
data (Eq a) => Conj a = C [a] deriving Show
```

É necessário que se consiga fazer o teste de pertença.

```
instance (Eq a) => OrdParcial (Conj a) where
  comp (C u) (C v) = let p1 = u `contido` v
                      p2 = v `contido` u
                      in if p1 && p2 then Just EQ else
                          if p1 then Just LT else
                          if p2 then Just GT
                          else Nothing
  where
    contido :: (Eq a) => [a] -> [a] -> Bool
    contido xs ys = all (\x-> elem x ys) xs
```

```
> (C [2,1]) `gt` (C [7,1,5,2])
Just False
> (C [2,1,3]) `lt` (C [7,1,5])
Nothing
```

```
> (C [2,1,2,1]) `lt` (C [7,1,5,5,2])
Just True
> (C [3,3,5,1]) `eq` (C [5,1,5,3,1])
Just True
```

141

A noção de *função finita* estabelece um conjunto de associações entre *chaves* e *valores*, para um conjunto finito de chaves.

Exemplo: Podemos agrupar numa *classe de construtores de tipos* as operações que devem estar definidas sobre funções finitas.

```
class Ffinita ff where
  val :: (Eq a) => a -> (ff a b) -> Maybe b
  acr :: (Eq a) => (a,b) -> (ff a b) -> (ff a b)
  def :: (Eq a) => a -> (ff a b) -> Bool
  dom :: (Eq a) => (ff a b) -> [a]

  def x t = case (val x t) of
    Nothing -> False
    (Just _) -> True
```

Exemplo: Tabelas implementando listas de associações (chave,valor) podem ser declaradas como instância da classe *Ffinita*.

```
data (Eq a) => Tab a b = Tab [(a,b)]
  deriving Show
```

É possível usar o mesmo nome para o *construtor de tipo* e para o *construtor de valores*.

142

```
instance Ffinita Tab where
  val x (Tab []) = Nothing
  val x (Tab ((c,v):xs)) = if x==c then Just v
                           else val x (Tab xs)

  acr (x,y) (Tab []) = Tab [(x,y)]
  acr (x,y) (Tab ((c,v):t)) = if x==c
                              then Tab ((x,y):t)
                              else let (Tab w) = acr (x,y) (Tab t)
                                   in Tab ((c,v):w)

  dom (Tab t) = map fst t
```

Exercício:

- Defina um tipo de dados polimórfico que implemente listas de associações em árvores binárias e que possa ser instância da classe *Ffinita*.
- Declare o construtor do tipo que acabou de definir como instância da classe *Ffinita*.

143

Mónades

Na programação funcional, conceito de *mónade* é usado para sintetizar a ideia de *computação*.

Uma *computação* é vista como algo que se passa dentro de uma *“caixa negra”* e da qual conseguimos apenas ver os resultados.

Em Haskell, o conceito de *mónade* está definido como uma classe de construtores de tipos.

```
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b -- “bind”
  (>>) :: m a -> m b -> m b -- “sequence”
  fail :: String -> m a

-- Minimal complete definition: (>>=), return
p >> q = p >>= \ _ -> q
fail s = error s
```

- O termo *(return x)* corresponde a uma computação nula que retorna o valor *x*.
- O operador *(>>=)* corresponde de alguma forma à composição de computações.

144