Programação Funcional — BCC222

Aulas 13,14

Tipos Abstratos
Tipos Algébricos
Classes de Tipos

Lucília Camarão de Figueiredo
Departamento de Ciência da Computação
Universidade Federal de Ouro Preto

Parte I

Tipos Abstratos

Conjuntos implementados como listas

Conjuntos como listas (1)

```
module SetList(Set,element,nil,insert,delete,
   set,allSet,subset,equal,showSet) where
```

Conjuntos como listas (1)

```
module SetList(Set,element,nil,insert,delete,
   set,allSet,subset,equal,showSet) where

data Set a = MkSet [a]

element ::  Eq a => a -> Set a -> Bool
element x (MkSet xs) = elem x xs

nil ::  Set a
nil = MkSet []

insert ::  a -> Set a -> Set a
insert x (MkSet xs) = MkSet (x:xs)

delete ::  Eq a => a -> Set a -> Set a
delete x (MkSet xs) = MkSet [ y | y <- xs, x /= y ]
```

Conjuntos como listas (2)

```
set ::  [a] -> Set a
set xs = MkSet xs

allSet ::  (a -> Bool) -> Set a -> Bool
allSet p (MkSet xs) = all p xs

subset ::  Eq a => Set a -> Set a -> Bool
s `subset` t = allSet (\x -> element x t) s

equal ::  Eq a => Set a -> Set a -> Bool
s `equal` t = s `subset` t && t `subset` s

showSet ::  Set a -> String
showSet (MkSet xs) = "MkSet " ++ show xs
```

Conjuntos como listas — uso (1)

```
module MainList where
import Test.QuickCheck
import SetList

s0 ::  Set Int
s0 = set [2,7,1,8,2,8]

prop_show ::  Bool
prop_show =
   showSet s0 ==
     "MkSet [2,7,1,8,2,8]"
```

Conjuntos como listas — uso (2)

```
prop_insert ::  Int -> Int -> [Int] -> Bool
prop_insert x y xs =
   element x (insert y s) == (x == y || element x s)
     where s = set xs

prop_delete ::  Int -> Int -> [Int] -> Bool
prop_delete x y xs =
   element x (delete y s) == (element x s && x /= y)
     where s = set xs

prop_insert_delete ::  Int -> [Int] -> Property
prop_insert_delete x xs =
   not (element x s) ==> s `equal` delete x (insert x s)
     where s = set xs
```

Conjuntos como listas — execução (1)

```
lucilia:  ghci MainList.hs
GHCi, version 6.10.4:  http://www.haskell.org/ghc/ :?  for he
Loading package ghc-prim ...  linking ...  done.
Loading package integer ...  linking ...  done.
Loading package base ...  linking ...  done.
[1 of 2] Compiling SetList ( SetList.hs, interpreted )
[2 of 2] Compiling MainList ( MainList.hs, interpreted )
Ok, modules loaded:  MainList, SetList.
MainList> element 2 s0
True
MainList> element 3 s0
False
MainList> s0
No instance for (Show (Set Int))
MainList> head s0
<interactive>:1:5:
Couldn't match expected type `[a]' against inferred type `Set a'
```

Conjuntos como listas — execução (1)

```
MainList> showSet s0
"MkSet [2,7,1,8,2,8]"
MainList> quickCheck prop_insert
OK, passed 100 tests.
MainList> quickCheck prop_delete
OK, passed 100 tests.
MainList> quickCheck prop_insert_delete
OK, passed 100 tests.
```

Parte II

Tipos Abstratos

Conjuntos implementados como árvores

Conjuntos como árvores (1)

```
module SetTree(Set,element,nil,insert,delete,
   set,allSet,subset,equal,showSet) where

data Set a = Nil | Node (Set a) a (Set a)

element ::  Ord a => a -> Set a -> Bool
element x Nil = False
element x (Node l y r)
    | x == y = True
    | x < y  = element x l
    | x > y  = element x r
```

Conjuntos como árvores (2)

```
nil ::  Set a
nil = Nil

insert ::  Ord a => a -> Set a -> Set a
insert x Nil = Node Nil x Nil
insert x (Node l y r)
    | x == y = Node l y r
    | x < y  = Node (insert x l) y r
    | x > y  = Node l y (insert x r)
```

Conjuntos como árvores (3)

```
set ::  Ord a => [a] -> Set a
set = foldr insert nil

allSet ::  (a -> Bool) -> Set a -> Bool
allSet p s = all p (list s)
   where list ::  Set a -> [a]
         list Nil          = []
         list (Node l x r) = list l ++ [x] ++ list r

subset ::  Ord a => Set a -> Set a -> Bool
s `subset` t = allSet (\x -> element x t) s

equal ::  Ord a => Set a -> Set a -> Bool
s `equal` t = s `subset` t && t `subset` s
```

Conjuntos como árvores (3)

```
showSet ::  Show a => Set a -> String
showSet Nil          = "Nil"
showSet (Node l x r) = "(Node " ++ showSet l
                       ++ " " ++ show x
                       ++ " " ++ showSet r ++ ")"
```

Conjuntos como árvores (5)

```
delete :: Ord a => a -> Set a -> Set a
delete x Nil = Nil
delete x (Node l y r)
   | x == y = join l r
   | x < y  = Node (delete x l) y r
   | x > y  = Node l y (delete x r)

join :: Ord a => Set a -> Set a -> Set a
join l Nil = l
join l r   = Node l x (delete x r)
   where x = smallest r

smallest :: Ord a => Set a -> a
smallest (Node Nil y r) = y
smallest (Node l y r)   = smallest l
```

Conjuntos como árvores — uso (1)

```
module MainTree where
import Test.QuickCheck
import SetTree

s0 ::  Set Int
s0 = set [2,7,1,8,2,8]

prop_show ::  Bool
prop_show =
  showSet s0 ==
    "(Node (Node (Node Nil 1 Nil) 2 (Node Nil 7 Nil)) 8 Nil)"

- set [2,7,1,8,2,8] ==
- insert 2 (insert 7 (insert 1
- insert 8 (insert 2 (insert 8 nil))))
```

Conjuntos como árvores — uso (2)

```
prop_insert ::  Int -> Int -> [Int] -> Bool
prop_insert x y xs =
  element x (insert y s) == (x == y || element x s)
     where s = set xs

prop_delete ::  Int -> Int -> [Int] -> Bool

prop_delete x y xs =
  element x (delete y s) == (element x s && x /= y)
     where s = set xs

prop_insert_delete ::  Int -> [Int] -> Property
prop_insert_delete x xs =
  not (element x s) ==> equalSet s (delete x (insert x s))
     where s = set xs
```

Conjuntos como árvores — execução (1)

```
lucilia:  ghci MainList.hs
GHCi, version 6.10.4:  http://www.haskell.org/ghc/ :?  for he
Loading package ghc-prim ...  linking ...  done.
Loading package integer ...  linking ...  done.
Loading package base ...  linking ...  done.
[1 of 2] Compiling SetList ( SetTree.hs, interpreted )
[2 of 2] Compiling MainList ( MainTree.hs, interpreted )
Ok, modules loaded:  MainList, SetTree.
MainTree> element 2 s0
True
MainTree> element 3 s0
False
MainTree> s0
No instance for (Show (Set Int))
```

Conjuntos como árvores — execução (2)

```
MainTree> showSet s0
"(Node (Node (Node Nil 1 Nil) 2 (Node Nil 7 Nil)) 8 Nil)"
```

```
(Node
  (Node
    (Node Nil 1 Nil)
    2
    (Node Nil 7 Nil))
  8
  Nil)"
```

Conjuntos como árvores — execução (3)

```
MainTree> quickCheck prop_insert
OK, passed 100 tests.
MainTree> quickCheck prop_delete
OK, passed 100 tests.
MainTree> quickCheck prop_insert_delete
OK, passed 100 tests.
```

Parte III

Classes de Tipos

Element

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

-- list comprehension
elem x ys = or [ x == y | y <- ys ]

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

-- ordem superior
elem x ys = foldr (||) False (map (x ==) ys)
```

Usando Element

```
Main> elem 1 [2,3,4]
False
Main> elem 'o' "word"
True
Main> elem (1,'o') [(0,'w'),(1,'o'),(2,'r'),(3,'d')]
True
Main> elem "word" ["list","of","word"]
True
Main> elem (\x -> x) [(\x -> -x), (\x -> -(-x))]
No instance for (Eq (a -> a))
```

Type class Eq

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

instance Eq Int where
  (==) = eqInt

instance Eq Char where
  x == y = ord x == ord y

instance (Eq a, Eq b) => Eq (a,b) where
  (u,v) == (x,y) = (u == x) && (v == y)

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

Element — tradução

```
data EqDict a = EqD (a -> a -> Bool)

eq ::  EqDict a -> a -> a -> Bool
eq (EqDict f) = f

elem ::  EqD a -> a -> [a] -> Bool

-- list comprehension
elem d x ys = or [ eq d x y | y <- ys ]

-- recursão
elem d x [] = False
elem d x (y:ys) = eq d x y || elem x ys

-- ordem superior
elem d x ys = foldr (||) False (map (eq d x) ys)
```

Type Classes — tradução

```
dInt ::  EqDict Int
dInt = EqD eqInt

dChar ::  EqDict Char
dChar = EqD f
   where f x y = eq dInt (ord x) (ord y)

dPair ::  (EqDict a, EqDict b) -> EqDict (a,b)
dPair (da,db) = EqD f
   where f (u,v) (x,y) = eq da u x && eq db v y

dList ::  EqDict a -> EqDict [a]
dList d = EqD f
   where f []     []     = True
         f []     (y:ys) = False
         f (x:xs) []     = False
         f (x:xs) (y:ys) = eq d x y && eq (dList d) xs ys
```

Usando element — tradução

```
Main> elem dInt 1 [2,3,4]
False
Main> elem dChar 'o' "word"
True
Main> elem (dPair dInt dChar) (1,'o') [(0,'w'),(1,'o')]
True
Main> elem (dList dChar) "word" ["list","of","word"]
True
```

Haskell usa tipos para escrever código para você!

Parte IV

Boolean

Type classes

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

class (Eq a) => Ord a where
  (<)  ::  a -> a -> Bool
  (<=) ::  a -> a -> Bool
  (>)  ::  a -> a -> Bool
  (>=) ::  a -> a -> Bool
  x <= y = x < y || x == y
  x >  y = y < x
  x >= y = y <= x

class Show a where
  show ::  a -> String
```

Instâncias para Boolean

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

instance Ord Bool where
  False < True  = True
  _     < _     = False

instance Show Bool where
  show False = "False"
  show True  = "True"
```

Boolean com cláusula deriving

```
data Bool = False | True
            deriving (Eq, Ord, Show)
```

Haskell usa tipos para escrever código para você!

Parte V

Conjuntos, de novo

Conjuntos como árvores, de novo

```
data Set a = Nil | Node (Set a) a (Set a)

instance Ord a => Eq (Set a) where
  s == t = s `subset` t && t `subset` s

instance Show a => Show (Set a) where
  show Nil          = "Nil"
  show (Node l x r) = "(Node " ++ show l
                    ++ " " ++ show x
                    ++ " " ++ show r ++ ")"
```

Conjuntos como árvores, de novo, de novo

```
data Set a = Nil | Node (Set a) a (Set a)
                deriving (Show)

instance Ord a => Eq (Set a) where
  s == t = s `subset` t && t `subset` s
```

Parte VI

Tuplas e Listas

Tuplas

```
data (a,b) = (a,b) deriving (Eq,Ord,Show)

instance (Eq a, Eq b) => Eq (a,b) where
  (x,y) == (x',y') = x == x' && y == y'

instance (Ord a, Ord b) => Ord (a,b) where
  (x,y) < (x',y') = x < x' || (x == x' && y < y')

instance (Show a, Show b) => Show (a,b) where
  show (x,y) = "(" ++ show x ++ "," ++ show y ++ ")"
```

Listas

```
data [a] = [] | a:[a] deriving (Eq,Ord,Show)

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

instance Ord a => Ord [a] where
   []     < []     = False
   []     < (y:ys) = True
   (x:xs) < []     = False
   (x:xs) < (y:ys) = x < y || (x == y && xs < ys)

instance Show a => Show [a] where
   show []     = "[]"
   show (x:xs) = "[" ++ showSep x xs ++ "]"
      where showSep x []       = show x
            showSep x (y:ys)   = show x ++ "," ++ showSep y ys
```

Parte VII

Igualdade sobre funções

Igualdade sobre funções

```
class Small a where
  forAll ::  (a -> Bool) -> Bool

instance Small Char where
  forAll p = and [ p x | x <- ['\000'..'\255'] ]

instance Small Bool where
  forAll p = and [ p x | x <- [False, True] ]

instance (Small a, Small b) => Small (a,b) where
  forAll p = forAll (\x -> (forAll (\y -> p (x,y))))

instance (Small a, Eq b) => Eq (a -> b) where
  f == g = forAll (\x -> f x == g x)

Main> elem (\x -> x) [(\x -> not x), (\x -> not (not x))]
True
```

Parte VIII

Seasons

Season

```
data Season = Winter | Spring | Summer | Fall

next ::  Season -> Season
next Winter = Spring
next Spring = Summer
next Summer = Fall
next Fall   = Winter

warm ::  Season -> Bool
warm Winter = False
warm Spring = True
warm Summer = True
warm Fall   = True
```

Season — cláusula deriving

```haskell
data Season = Winter | Spring | Summer | Fall
              deriving (Eq, Ord, Show, Enum)
```

```haskell
instance Eq Seasons where
  Winter == Winter = True
  Spring == Spring = True
  Summer == Summer = True
  Fall   == Fall   = True
  _      == _      = False

instance Ord Seasons where
  Winter < Spring = True
  Winter < Summer = True
  Winter < Fall   = True
  Spring < Summer = True
  Spring < Fall   = True
  Summer < Fall   = True
  _      < _      = False

instance Show Seasons where
  show Winter = "Winter"
  show Spring = "Spring"
  show Summer = "Summer"
  show Fall   = "Fall"
```

Classe Enum

```
class Enum a where
  succ, pred       ::  a -> a
  toEnum           ::  Int -> a
  fromEnum         ::  a -> Int
  enumFrom         ::  a -> [a]                -- [x..]
  enumFromTo       ::  a -> a -> [a]           -- [x..y]
  enumFromThen     ::  a -> a -> [a]           -- [x,y..]
  enumFromThenTo   ::  a -> a -> a -> [a]  -- [x,y..z]

  succ x           = toEnum (fromEnum x + 1)
  pred x           = toEnum (fromEnum x - 1)
  enumFrom x
     = map toEnum [fromEnum x ..]
  enumFromTo x y
     = map toEnum [fromEnum x ..  fromEnum y]
  enumFromThen x y
     = map toEnum [fromEnum x, fromEnum y ..]
  enumFromThenTo x y z
     = map toEnum [fromEnum x, fromEnum y ..  fromEnum z]
```

Açúcar sintático

```
-- [x..]    = enumFrom x
-- [x..y]   = enumFromTo x y
-- [x,y..]  = enumFromThen x y
-- [x,y..z] = enumFromThenTo x y z

instance Enum Int where
  toEnum x          = x
  fromEnum x        = x
  succ x            = x+1
  pred x            = x-1
  enumFrom x        = iterate (+1) x
  enumFromTo x y    = takeWhile (<= y) (iterate (+1) x)
  enumFromThen x y  = iterate (+(y-x)) x
  enumFromThenTo x y z
                    = takeWhile (<= z) (iterate (+(y-x)) x)

iterate ::  (a -> a) -> a -> [a]
iterate f x = x :  iterate f (f x)
```

Instância derivada automaticamente

```
instance Enum Seasons where
  fromEnum Winter = 0
  fromEnum Spring = 1
  fromEnum Summer = 2
  fromEnum Fall   = 3

  toEnum 0 = Winter
  toEnum 1 = Spring
  toEnum 2 = Summer
  toEnum 3 = Fall
```

Season, de novo

```
next ::  Season -> Season
next x = toEnum ((fromEnum x + 1) `mod` 4)

warm ::  Season -> Bool
warm x = x `elem` [Spring..Fall]
-- [Spring..Fall] = [Spring, Summer, Fall]
```

Parte IX

Shape

Shape

```
type Radius = Float
type Width  = Float
type Height = Float

data Shape = Circle Radius
           | Rect Width Height
           deriving (Eq, Ord, Show)

area ::  Shape -> Float
area (Circle r) = pi * r^2
area (Rect w h) = w * h
```

Instâncias derivadas

```
instance Eq Shape where
  Circle r == Circle r'   = r == r'
  Rect a b == Rect a' b'  = a == a' && b == b'
  _        == _           = False

instance Ord Shape where
  Circle r < Circle r'    = r < r'
  Rect a b < Rect a' b'   = a < a' || (a == a') && b < b'
  Circle _ < Rect _ _     = True
  Rect _ _ < Circle _     = False

instance Show Shape where
  show (Circle r) = "Circle " ++ show r
  show (Rect a b) = "Rect " ++ show a ++  ++ show b
```

Parte X

Expressões

Árvores de Expressões

```
data Exp = Lit Int
         | Exp :+: Exp
         | Exp :*: Exp
         deriving (Eq, Ord, Show)

eval ::  Exp -> Int
eval (Lit n)     = n
eval (e :+:  f) = eval e + eval f
eval (e :*:  f) = eval e * eval f

Main> eval (Lit 2 :+:  (Lit 3 :*:  Lit 3))
11
Main> eval ((Lit 2 :+:  Lit 3) :*:  Lit 3)
15
```

Instâncias derivadas

```
instance Eq Exp where
  Lit n     == Lit n'      = n == n'
  e :+: f   == e' :+: f'   = e == e' && f == f'
  e :*: f   == e' :*: f'   = e == e' && f == f'
  _         == _           = False

instance Ord Exp where
  Lit n     < Lit n'      = n < n'
  Lit n     < e' :+: f'   = True
  Lit n     < e' :*: f'   = True
  e :+: f   < e' :+: f'   = e < e' || (e == e' && f < f')
  e :+: f   < e' :*: f'   = True
  e :*: f   < e' :*: f'   = e < e' || (e == e' && f < f')

instance Show Exp where
  show (Lit n)     = "Lit " ++ showN n
  show (e :+: f)   = "(" ++ show e ++ ":+:" ++ show f ++ ")"
  show (e :*: f)   = "(" ++ show e ++ ":*:" ++ show f ++ ")"
```

Parte XI

Números

Classes numéricas

```
class (Eq a, Show a) => Num a where
  (+),(-),(*)   ::  a -> a -> a
  negate        ::  a -> a
  fromInteger   ::  Integer -> a

class (Num a) => Fractional a where
  (/) :         ::  a -> a -> a
  recip         ::  a -> a
  fromRational  ::  Rational -> a

  recip x       = 1/x

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

class (Real a, Enum a) => Integral a where
  div, mod      ::  a -> a -> a
  toInteger     ::  a -> Integer
```

Um tipo numérico pré-definido

```
instance Num Float where
  (+)          = builtInAddFloat
  (-)          = builtInSubtractFloat
  (*)          = builtInMultiplyFloat
  negate       = builtInNegateFloat
  fromInteger  = builtInFromIntegerFloat

instance Fractional Float where
  (/)          = builtInDivideFloat
  fromRational = builtInFromRationalFloat
```

Um tipo numérico definido pelo usuário

```
data Point = Pnt Float Float

scalar ::  Float -> Point
scalar x = Pnt x x

instance Num Point where
  Pnt x y + Pnt x' y' = Pnt (x+x') (y+y')
  Pnt x y - Pnt x' y' = Pnt (x-x') (y-y')
  Pnt x y * Pnt x' y' = Pnt (x*x') (y*y')
  negate (Pnt x y)     = Pnt (-x) (-y)
  fromInteger z        = scalar (fromInteger z)

instance Fractional Point where
  Pnt x y / Pnt x' y' = Pnt (x/x') (y/y')
  fromRational z       = scalar (fromRational z)
```

Pontos

```
instance Eq Point where
  Pnt x y == Pnt x' y'      = x == x' && y == y'

instance Ord Point where
  Pnt x y < Pnt x' y'       = x < x' && y < y'

glb,lub ::  Point -> Point -> Point
Pnt x y `glb` Pnt x' y'    = Pnt (x `min` x') (y `min` y')
Pnt x y `lub` Pnt x' y'    = Pnt (x `max` x') (y `max` y')
```