Programare funcțională

Clase de tipuri

Ioana Leuștean
Traian Șerbănuță

Departamentul de Informatică, FMI, UB

Exemplu: test de apartenență

Să scriem funcția **elem** care testează dacă un element aparține unei liste.

- definiția folosind descrieri de liste

Exemplu: test de apartenență

Să scriem funcția **elem** care testează dacă un element aparține unei liste.

- definiția folosind descrieri de liste

```
elem x ys = or [ x == y | y <- ys ]
```

Exemplu: test de apartenență

Să scriem funcția **elem** care testează dacă un element aparține unei liste.

- definiția folosind descrieri de liste

```
elem x ys       = or [ x == y | y <- ys ]
```

- definiția folosind recursivitate

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

Exemplu: test de apartenență

Să scriem funcția **elem** care testează dacă un element aparține unei liste.

- definiția folosind descrieri de liste

```
elem x ys       = or [ x == y | y <- ys ]
```

- definiția folosind recursivitate

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

- definiția folosind funcții de nivel înalt

```
elem x ys       = foldr (||) False (map (x ==) ys)
```

Funcția elem este polimorfică

```
*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
```

Funcția elem este polimorfică

```
*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
```

Care este tipul funcției **elem**?

Funcția elem este polimorfică

```
*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
```

Care este tipul funcției **elem**?

Funcția **elem** este polimorfică.

Definiția funcției este parametrică în tipul de date.

Funcția elem este polimorfică

Dar nu pentru orice tip

- Totuși definiția nu funcționează pentru orice tip!

```
*Main> elem (+ 2) [(+ 2), sqrt]
No instance for (Eq (Double -> Double)) arising from a use of ‘elem‘
```

Ce se întâmplă?

Funcția elem este polimorfică

Dar nu pentru orice tip

- Totuși definiția nu funcționează pentru orice tip!

```
*Main> elem (+ 2) [(+ 2), sqrt]
```
No instance for (Eq (Double -> Double)) arising from a use of ‘elem‘

Ce se întâmplă?

```
> :t elem_
elem_ :: Eq a => a -> [a] -> Bool
```

În definiția

```
elem x ys       = or [ x == y | y <- ys ]
```

folosim relația de egalitate == care nu este definită pentru orice tip:

```
Prelude> sqrt == sqrt
```
No instance for (Eq (Double -> Double)) ...
```
Prelude> ("ab",1) == ("ab",2)
False
```

Clase de tipuri

- O clasă de tipuri este determinată de o mulțime de funcții (este o interfață).

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  -- minimum definition: (==)
  x /= y = not (x == y)
  -- ^^^ putem avea definitii implicite
```

Clase de tipuri

- O clasă de tipuri este determinată de o mulțime de funcții (este o interfață).

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  -- minimum definition: (==)
  x /= y = not (x == y)
  -- ^^^ putem avea definitii implicite
```

- Tipurile care aparțin clasei sunt instanțe ale clasei.

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

Clasa de tipuri. Constrângeri de tip

- În signatura funcției **elem** trebuie să precizăm ca tipul a este în clasa **Eq**

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

Eq a se numește constrângere de tip. **=>** separă constrângerile de tip de restul signaturii.

Clasa de tipuri. Constrângeri de tip

- În signatura funcției **elem** trebuie să precizăm ca tipul a este în clasa **Eq**

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

Eq a se numește constrângere de tip. **=>** separă constrângerile de tip de restul signaturii.

- În exemplul de mai sus am considerat că **elem** este definită pe liste, dar în realitate funcția este mai complexă

```
Prelude> :t elem
elem :: (Eq a, Foldable t) => a -> t a -> Bool
```

În această definiție Foldable este o altă clasă de tipuri, iar t este un parametru care ține locul unui *constructor de tip*!

Clasa de tipuri. Constrângeri de tip

- În signatura funcției **elem** trebuie să precizăm ca tipul a este în clasa **Eq**

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

Eq a se numește constrângere de tip. **=>** separă constrângerile de tip de restul signaturii.

- În exemplul de mai sus am considerat că **elem** este definită pe liste, dar în realitate funcția este mai complexă

```
Prelude> :t elem
elem :: (Eq a, Foldable t) => a -> t a -> Bool
```

În această definiție Foldable este o altă clasă de tipuri, iar t este un parametru care ține locul unui *constructor de tip*!

Sistemul tipurilor in Haskell este complex!

Instanțe ale lui **Eq**

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

instance Eq Int       where
  (==) = eqInt    -- built-in

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

Instanțe ale lui **Eq**

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

instance Eq Int       where
  (==) = eqInt   -- built-in

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)
```

Eq, Ord

- Clasele pot fi extinse

```
class (Eq a) => Ord a where
  (<)  :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  (>)  :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  -- minimum definition: (<=)
  x < y  = x <= y && x /= y
  x > y  = y < x
  x >= y = y <= x
```

Eq, Ord

- Clasele pot fi extinse

```
class (Eq a) => Ord a where
  (<)  :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  (>)  :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  -- minimum definition: (<=)
  x < y  = x <= y && x /= y
  x > y  = y < x
  x >= y = y <= x
```

Clasa **Ord** este clasa tipurilor de date înzestrate cu o relație de ordine.

În definiția clasei **Ord** s-a impus o constrângere de tip. Astfel, orice instanță a clasei **Ord** trebuie să fie instanță a clasei **Eq**.

Instanțe ale lui **Ord**

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

Instanțe ale lui **Ord**

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

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

Instanțe ale lui **Ord**

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

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

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

Definirea claselor

Să presupunem că vrem să definim o clasă de tipuri pentru datele care pot fi afișate.

Definirea claselor

Să presupunem că vrem să definim o clasă de tipuri pentru datele care pot fi afișate. O astfel de clasă trebuie să conțină o metodă care să indice modul de afișare:

Definirea claselor

Să presupunem că vrem să definim o clasă de tipuri pentru datele care pot fi afișate. O astfel de clasă trebuie să conțină o metodă care să indice modul de afișare:

```
class Visible a where
    toString :: a -> String
```

Definirea claselor

Să presupunem că vrem să definim o clasă de tipuri pentru datele care pot fi afișate. O astfel de clasă trebuie să conțină o metodă care să indice modul de afișare:

```
class Visible a where
    toString :: a -> String
```

Putem face instanțieri astfel:

```
instance Visible Char where
    toString c = [c]
```

Definirea claselor

Să presupunem că vrem să definim o clasă de tipuri pentru datele care pot fi afișate. O astfel de clasă trebuie să conțină o metodă care să indice modul de afișare:

```
class Visible a where
    toString :: a -> String
```

Putem face instanțieri astfel:

```
instance Visible Char where
    toString c = [c]
```

Clasele **Eq**, **Ord** sunt predefinite. Clasa Visible este definită de noi, dar există o clasă predefinită care are același rol: clasa **Show**

Show

```
class Show a where
  show :: a -> String      -- analogul lui "toString"
```

Show

```
class Show a where
  show :: a -> String      -- analogul lui "toString"

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

Show

```
class Show a where
  show :: a -> String     -- analogul lui "toString"

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

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

Show

```
class Show a where
  show :: a -> String     -- analogul lui "toString"

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

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

  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
```

Clase de tipuri pentru numere

```
class (Eq a, Show a) => Num a where
  (+),(-),(*)     :: a -> a -> a
  negate          :: a -> a
  fromInteger     :: Integer -> a
  -- minimum definition: (+),(-),(*),fromInteger
  negate x        =   fromInteger 0 - x
```

Clase de tipuri pentru numere

```
class (Eq a, Show a) => Num a where
  (+),(-),(*)    :: a -> a -> a
  negate         :: a -> a
  fromInteger    :: Integer -> a
  -- minimum definition: (+),(-),(*),fromInteger
  negate x       =  fromInteger 0 - x

class (Num a) => Fractional a where
  (/)            :: a -> a -> a
  recip          :: a -> a
  fromRational   :: Rational -> a
  -- minimum definition: (/), fromRational
  recip x        =  1/x
```

Clase de tipuri pentru numere

```
class (Eq a, Show a) => Num a where
  (+),(-),(*)    :: a -> a -> a
  negate         :: a -> a
  fromInteger    :: Integer -> a
  -- minimum definition: (+),(-),(*),fromInteger
  negate x       =  fromInteger 0 - x

class (Num a) => Fractional a where
  (/)            :: a -> a -> a
  recip          :: a -> a
  fromRational   :: Rational -> a
  -- minimum definition: (/), fromRational
  recip x        =  1/x

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

Clase de tipuri pentru numere

```
class (Eq a, Show a) => Num a where
  (+),(-),(*)    :: a -> a -> a
  negate         :: a -> a
  fromInteger    :: Integer -> a
  -- minimum definition: (+),(-),(*),fromInteger
  negate x       =  fromInteger 0 - x

class (Num a) => Fractional a where
  (/)            :: a -> a -> a
  recip          :: a -> a
  fromRational   :: Rational -> a
  -- minimum definition: (/), fromRational
  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
```

Clasa de tipuri Enum

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

Int ca instanță a lui Enum

```
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)

takeWhile :: (a -> Bool) -> [a]    ->    [a]
takeWhile p []                     =     []
takeWhile p (x:xs) | p x           =     x : takeWhile p xs
                   | otherwise     =     []
```

Anotimpuri

```
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
```

Anotimpuri — Instanțiere manuala pentru Eq, Ord, Show

```
instance Eq Seasons where
  Winter == Winter = True
  Spring == Spring = True
  Summer == Summer = True
  Fall   == Fall   = True
  _      == _      = False
instance Ord Seasons where
  Spring <= Winter = False
  Summer <= Winter = False
  Summer <= Spring = False
  Fall   <= Winter = False
  Fall   <= Spring = False
  Fall   <= Summer = False
  _      <= _      = True
instance Show Seasons where
  show Winter = "Winter"
  show Spring = "Spring"
  show Summer = "Summer"
  show Fall   = "Fall"
```

Instanță Enum pentru anotimpuri

```
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
```

Anotimpuri folosind derivare automată

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

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

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

Toate funcțiile pentru clasele derivate sunt generate automat!

Pe săptămâna viitoare!