Programare funcțională

Evaluare Lenesă. Clase de tipuri. Date structurate.

Ioana Leuștean
Traian Florin Șerbănuță

Departamentul de Informatică, FMI, UB
ioana@fmi.unibuc.ro
traian.serbanuta@unibuc.ro

1. Evaluare lenesă. Memoizare

2. Clase de tipuri

Evaluare lenesă. Memoizare

Evaluare lenesă. Liste infinite

- Putem folosi funcțiile **map** și **filter** pe liste infinite:

```
Prelude> inf = map (+10) [1..] -- inf nu este evaluat
Prelude> take 3 inf
[11,12,13]
```

Limbajul Haskell folosește implicit evaluarea lenesă

- expresiile sunt evaluate numai când este nevoie de valoarea lor
- expresiile nu sunt evaluate total, elementele care nu sunt folosite rămân neevaluate
- o expresie este evaluată o singură dată.

Evaluare lenesă. Liste infinite

- Putem folosi funcțile **map** și **filter** pe liste infinite:

```
Prelude> inf = map (+10) [1..] -- inf nu este evaluat
Prelude> take 3 inf
[11,12,13]
```

Limbajul Haskell folosește implicit evaluarea lenesă

- expresiile sunt evaluate numai când este nevoie de valoarea lor
- expresiile nu sunt evaluate total, elementele care nu sunt folosite rămân neevaluate
- o expresie este evaluată o singură dată.

În exemplul de mai sus, este acceptată definiția lui `inf`, fără a fi evaluată. Când expresia **take** 3 `inf` este evaluată, numai primele 3 elemente ale lui `inf` sunt calculate, restul rămânând neevaluate.

Evaluare lenesă. Partajare termeni

```
take :: Int -> [a] -> [a]
take !_  []     = []
take 1   (x: _) = [x]
take m   (x:xs) = x : take (m - 1) xs

zipWith :: (a->b->c) -> [a]->[b]->[c]
zipWith f = go
  where go [] _ = []
        go _ [] = []
        go (x:xs) (y:ys) = f x y : go xs ys

fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

- Pattern-urile stricte (prefixate de **!** forțează evaluarea argumentului)

Evaluare lenesă. Partajare termeni

```
take :: Int -> [a] -> [a]
take !_  []     = []
take 1   (x: _) = [x]
take m   (x:xs) = x : take (m - 1) xs

zipWith :: (a->b->c) -> [a]->[b]->[c]
zipWith f = go
  where go [] _ = []
        go _ [] = []
        go (x:xs) (y:ys) = f x y : go xs ys

fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

- Pattern-urile stricte (prefixate de **!** forțează evaluarea argumentului)
- Aparițiile multiple ale aceleiași variabile într-o ecuație sunt identificate

Optimizarea recursiei: Memoizare

```
fib :: Int -> Integer
fib = f
  where
    f 0 = 0
    f 1 = 1
    f n = (genf !! (n-2)) + (genf !! (n-1))

    genf = map f [0..]
```

```
*Main> fib 200
280571172992510140037611932413038677189525
(0.01 secs, 206,448 bytes)
```

Important! genf e definită separat de f

Clase de tipuri

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
```