Programare declarativă

Introducere în programarea funcțională folosind Haskell

Traian Florin Șerbănuță - seria 33
Ioana Leuștean - seria 34

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

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

Funcția **elem** este polimorfică.

Definiția funcției este parametrică în tipul de date.

```
*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

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
```

- 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 constrăngerile 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 constrăngerile 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

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

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


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.
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"

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

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
```

Tipuri de date algebrice

Tipuri suma

În Haskell tipul **Bool** este definit astfel:

```
data Bool = False | True
```

Bool este un tip de date sumă, în care:

Bool este constructor de tip

False și **True** sunt constructori de date

- Se definesc operații:

```
not :: Bool -> Bool
not False = True
not True  = False

(&&), (||) :: Bool -> Bool -> Bool
False && q = False
True  && q = q
False || q = q
True  || q = True
```

Tip sumă: anotimpuri

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

instance Enum Season where
  succ Spring = Summer
  succ Summer = Autumn
  succ Autumn = Winter
  succ Winter = Spring

  fromEnum Winter = 0
  fromEnum Spring = 1
  fromEnum Summer = 2
  fromEnum Fall   = 3

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

```
class Enum a where
  succ :: a -> a
  fromEnum :: a -> Int
  toEnum :: Int -> a
```

Tipuri produs

Să definim un tip de date care să aibă ca valori "punctele" cu două coordonate de tipuri oarecare:

```
data Point a b = Pt a b
```

Point este un tip de date produs, în care:

Point este constructor de tip

Pt este constructor de date

- Se pot defini operații:

```
pointFlip :: Point a b -> Point b a
pointFlip (Pt x y) = Pt y x
```

Tipuri produs

Să definim un tip de date care să aibă ca valori "punctele" cu două coordonate de tipuri oarecare:

```
data Point a b = Pt a b
```

Point este un tip de date produs, în care:

Point este constructor de tip

Pt este constructor de date

- Se pot defini operații:

```
pointFlip :: Point a b -> Point b a
pointFlip (Pt x y) = Pt y x
```

- Pentru a accesa componentele, definim proiecțiile:

```
pr1 : Point a b -> a
pr1 (Pt x _) = x
pr2 : Point a b -> b
pr2 (Pt _ y) = y
```

Tipuri de date algebrice

Forma generală

$$\begin{aligned} data\ \ Typename \quad = \quad & Cons_1 \ \ t_{11} \ldots t_{1k_1} \\ & \mid Cons_2 \ \ t_{21} \ldots t_{2k_2} \\ & \mid \ldots \\ & \mid Cons_n \ \ t_{n1} \ldots t_{nk_n} \end{aligned}$$

unde $k_1, \ldots, k_n \geq 0$

- Se pot folosi tipuri sumă și tipuri produs.
- Se pot defini tipuri parametrizate.
- Se pot folosi definiții recursive.

Atenție! Alternativele trebuie să conțină constructori.

data StrInt = **String** | **Int** este greșit

data StrInt = VS **String** | VI **Int** este corect

[VI 1, VS "abc", VI 34, VI 0, VS "xyz"] :: [StrInt]

Tipuri de date algebrice - exemple

```
data Bool = False | True

data Season = Winter | Spring | Summer | Fall

data Shape = Circle Float | Rectangle Float Float
```

Tipuri de date algebrice - exemple

```
data Bool = False | True

data Season = Winter | Spring | Summer | Fall

data Shape = Circle Float | Rectangle Float Float

data Maybe a = Nothing | Just a

data Pair a b = Pair a b
    -- constructorul de tip si cel de date pot sa coincida
```

Tipuri de date algebrice - exemple

```
data Bool = False | True

data Season = Winter | Spring | Summer | Fall

data Shape = Circle Float | Rectangle Float Float

data Maybe a = Nothing | Just a

data Pair a b = Pair a b
  -- constructorul de tip si cel de date pot sa coincida

data Nat = Zero | Succ Nat

data Exp = Lit Int | Add Exp Exp | Mul Exp Exp

data List a = Nil | Cons a (List a)

data Tree a = Empty | Leaf a | Branch (Tree a) (Tree a)
```

Exemplu - date personale. Utilizarea **type**

Cu **type** se pot redenumi tipuri deja existente.

```
type FirstName = String
type LastName = String
type Age = Int
type Height = Float
type Phone = String

data Person = Person FirstName LastName Age Height Phone
```

Exemplu - date personale. Proiecții

```
firstName :: Person -> String
firstName (Person firstname _ _ _ _ _) = firstname

lastName :: Person -> String
lastName (Person _ lastname _ _ _ _) = lastname

age :: Person -> Int
age (Person _ _ age _ _ _) = age

height :: Person -> Float
height (Person _ _ _ height _ _) = height

phoneNumber :: Person -> String
phoneNumber (Person _ _ _ _ number _) = number
```

Exemplu - date personale. Utilizare

```
Main*> let ionel = Person "Ion" "Ionescu" 20 175.2  "
    0712334567"
Main*> firstName ionel
"Ion"
Main*> height ionel
175.2
Main*> phoneNumber ionel
"0712334567"
```

Date personale ca înregistrări

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     , height :: Float
                     , phoneNumber :: String
                     }
```

Date personale ca înregistrări

- Putem folosi atât forma algebrică cât și cea de înregistrare

```
ionel = Person "Ion" "Ionescu" 20 175.2 "0712334567"

gigel = Person { firstName = "Gheorghe"
               , lastName="Georgescu"
               , age = 30, height = 192.3
               , phoneNumber = "0798765432"
               }
```

- Putem folosi și pattern-matching
- Proiecțiile sunt definite automat; sintaxă specializată pentru actualizări

```
nextYear :: Person -> Person
nextYear person = person { age = age person + 1 }
```

Derivare automata pentru tipuri algebrice

Am definit tipuri de date noi:

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

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

Cum putem să le facem instanțe ale claselor **Eq**, **Ord**, **Show**?

Putem să le facem explicit sau să folosim derivarea automată.

Atenție!
Derivarea automată poate fi folosită numai pentru unele clase predefinite.

Derivare automata pentru tipuri algebrice

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

Egalitatea, relația de ordine și modalitatea de afișare sunt definite implicit dacă este posibil:

```
*Main> Pt 2 3 < Pt 5 6
True

*Main> Pt 2 "b" < Pt 2 "a"
False

*Main Data.Char> Pt (+2) 3 < Pt (+5) 6

<interactive>:69:1: error:• No instance for (Ord (Integer -> Integer))arising from a
      use of ‘<’
```

Instanțiere explicită - exemplu

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

eqSeason :: Season -> Season -> Bool
eqSeason Spring Spring = True
eqSeason Summer Summer = True
eqSeason Autumn Autumn = True
eqSeason Winter Winter = True
eqSeason _      _      = False


showSeason :: Season -> String
showSeason Spring = "Spring"
showSeason Summer = "Summer"
showSeason Autumn = "Autumn"
showSeason Winter = "Winter"
```

```
instance Eq Season where
    (==) = eqSeason
```

```
instance Show Season where
    show = showSeason
```

Exemplu: numerele naturale (Peano)

Declarație ca tip de date algebric folosind șabloane

```
data Nat = Zero | Succ Nat

(^^^) :: Float -> Nat -> Float
x ^^^ Zero     = 1.0
x ^^^ (Succ n) = x * x ^^^ n
```

Comparați cu versiunea folosind notația predefinită

```
(^^) :: Float -> Int -> Float
x ^^ 0 = 1.0
x ^^ n = x * (x ^^ (n-1))
```

Exemplu: adunare și înmulțire pe Nat

Definiție pe tipul de date algebric

```
(+++) :: Nat -> Nat -> Nat
m +++ Zero     = m
m +++ (Succ n) = Succ (m +++ n)

(***) :: Nat -> Nat -> Nat
m *** Zero     = Zero
m *** (Succ n) = (m *** n) +++ m
```

Comparați cu versiunea folosind notația predefinită

```
(+) :: Int -> Int -> Int
m + 0 = m
m + n = (m + (n-1)) + 1

(*) :: Int -> Int -> Int
m * 0 = 0
m * n = (m * (n-1)) + m
```

Exemplu: liste

Declarație ca tip de date algebric

```
data  List a  = Nil
              | Cons a (List a)

append :: List a -> List a -> List a
append Nil ys           = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

Constructori simboluri

Declarație ca tip de date algebric cu simboluri

```
data  List a  = Nil
              | a ::: List a
          deriving (Show)
infixr 5 :::

(+++) :: List a -> List a -> List a
infixr 5 +++
Nil +++ ys        = ys
(x ::: xs) +++ ys = x ::: (xs +++ ys)
```

Comparați cu versiunea folosind notația predefinită

```
(++) :: [a] -> [a] -> [a]
[] ++ ys      = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Definirea egalității și a reprezentării

Eq și Show

```
eqList :: Eq a => List a -> List a -> Bool
eqList Nil Nil = True
eqList (x ::: xs) (y:::ys) = x == y && eqList xs ys
eqList _ _ = False

showList :: Show a => List a -> String
showList Nil = "Nil"
showList (x ::: xs) = show x ++ " ::: " ++ showList xs

instance (Eq a) => Eq (List a) where
  (==) = eqList

instance (Show a) => Show (List a) where
  show = showList
```

Pe săptămâna viitoare!