Programare funcțională

Introducere în programarea funcțională folosind Haskell
C06

Claudia Chiriță
Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Înregistrări

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

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

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

```
Prelude> let ionel = Person "Ion" "Ionescu" 20 175.2
    "0712334567"

Prelude> firstName ionel
"Ion"

Prelude> height ionel
175.2

Prelude> phoneNumber ionel
"0712334567"
```


Date personale ca înregistrări

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

Date personale ca înregistrări

```
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 }
```


Date personale ca înregistrări

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     , height :: Float
                     , phoneNumber :: String
                     }
ionel = Person "Ion" "Ionescu" 20 175.2 "0712334567"

nextYear person = person { age = age person + 1 }
```

```
Prelude> nextYear ionel
No instance for (Show Person) arising from a use of 'print'
```

Deși toate definițiile sunt corecte, o valoare de tip Person nu poate fi afișată deoarece nu are o instanță a clasei **Show**.

Clase de tipuri

Exemplu: test de apartenență

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

- definiția folosind descrieri de liste

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

- definiția folosind recursivitate

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

- definiția folosind funcții de nivel înalt

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


Funcția elem este polimorfică

```
Prelude> my_elem 1 [2,3,4]
False

Prelude> my_elem 'o' "word"
True

Prelude> my_elem (1,'o') [(0,'w'),(1,'o'),(2,'r'),(3,'d
    ')]
True

Prelude> my_elem "word" ["list","of","word"]
True
```

Care este tipul funcției my_elem?

Funcția my_elem este polimorfică.

Definiția funcției este parametrică în tipul de date.

Funcția elem este polimorfică

Totuși definiția nu funcționează pentru orice tip!

```
Prelude> my_elem (+ 2) [(+ 2), sqrt]
No instance for (Eq (Double -> Double)) arising from a use of 'elem'
```

Ce se întâmplă?

```
Prelude> :t my_elem
my_elem :: Eq a => a -> [a] -> Bool
```

În definiția

```
my_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 my_elem trebuie să precizăm ca tipul a este în clasa **Eq**

```
my_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 my_elem este definită pe liste, dar în realitate funcția este mai complexă:

```
Prelude> :t my_elem
my_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
```

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')
  -- 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. 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
```


Clase de tipuri pentru numere

```
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
  ...
```

Puteti verifica folosind comanda :info sau :i ce conține o anumită clasă de tipuri.

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 vs Instanțiere explictă

O clasă de tipuri este determinată de o mulțime de funcții.

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

Tipurile care aparțin clasei sunt instanțe ale clasei.

Instanțierea prin derivare automată:

```
data Point a b = Pt a b
                 deriving Eq
```

Instanțiere explicită:

```
instance Eq a => Eq (Point a b) where
      (==) (Pt x1 y1) (Pt x2 y2) = (x == x1)
```


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:

```
Prelude> Pt 2 3 < Pt 5 6
True

Prelude> Pt 2 "b" < Pt 2 "a"
False

Prelude> Pt (+2) 3 < Pt (+5) 6
No instance for (Ord (Integer -> Integer)) arising from a use of '<'
```

Instanțiere explicită - exemplu

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

Instance Eq Season where
  Spring == Spring = True
  Summer == Summer = True
  Autumn == Autumn = True
  Winter == Winter = True
  _      ==    _   = False

Instance Show Season where
  show Spring = "Primavara"
  show Summer = "Vara"
  show Autumn = "Toamna"
  show Winter = "Iarna"
```


Exemplu: liste

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

Exemplu de operație:

```
(+++) :: 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)
```


Constructori simboluri

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

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


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

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


Pe săptămâna viitoare!