

Programare declarativă

Introducere în programarea funcțională folosind Haskell

Ioana Leuștean
Ana Cristina Turlea

Departamentul de Informatică, FMI, UB
ioana@fmi.unibuc.ro
ana.turlea@fmi.unibuc.ro

- 1 Parțialitate - tipul Maybe
- 2 Variante - tipul Either
- 3 Logică propozițională
- 4 Expresii
- 5 Arbori
- 6 Clasa Foldable

Tipuri de date algebrice

Tipurile de date algebrice se definesc folosind "operațiile" sumă și produs.

Forma generală

$$\begin{aligned} \text{data } \textit{Typename} \quad = \quad & \textit{Cons}_1 \ t_{11} \dots t_{1k_1} \\ & | \textit{Cons}_2 \ t_{21} \dots t_{2k_2} \\ & | \dots \\ & | \textit{Cons}_n \ t_{n1} \dots t_{nk_n} \end{aligned}$$

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

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

Derivare automata vs Instanțiere explicită

- 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)
```

Parțialitate - tipul Maybe

Tipul Maybe (opțiune)

```
data Maybe a = Nothing | Just a
```

Argumente opționale

```
power :: Maybe Int -> Int -> Int
power Nothing n    = 2 ^ n
power (Just m) n = m ^ n
```

Rezultate opționale

```
divide :: Int -> Int -> Maybe Int
divide n 0 = Nothing
divide n m = Just (n 'div' m)
```

Maybe - folosirea unui rezultat opțional

```
divide :: Int -> Int -> Maybe Int
divide n 0 = Nothing
divide n m = Just (n `div` m)
```

-- *utilizare gresita*

```
wrong :: Int -> Int -> Int
wrong n m = divide n m + 3
```

-- *utilizare corecta*

```
right :: Int -> Int -> Int
right n m = case divide n m of
    Nothing -> 3
    Just r -> r + 3
```

Variante - tipul Either

Either A B (A sau B)

```
data Either a b = Left a | Right b
```

```
mylist :: [Either Int String]
```

```
mylist = [Left 4, Left 1, Right "hello", Left 2,  
         Right " ", Right "world", Left 17]
```

Definiți o funcție care calculează suma elementelor întregi.

```
addints    :: [Either Int String] -> Int
```

```
addints    [] = 0
```

```
addints    (Left n : xs) = n + addints xs
```

```
addints    (Right s : xs) = addints xs
```

```
addints'   :: [Either Int String] -> Int
```

```
addints'   xs = sum [n | Left n <- xs]
```

A sau B

```
data Either a b = Left a | Right b
```

```
mylist :: [Either Int String]
mylist = [Left 4, Left 1, Right "hello", Left 2,
          Right " ", Right "world", Left 17]
```

Definiți o funcție care întoarce concatenarea elementelor de tip **String**.

```
addstrs    :: [Either Int String] -> String
addstrs    []                        = ""
addstrs    (Left n : xs)            = addstrs xs
addstrs    (Right s : xs) = s ++ addstrs xs
```

```
addstrs'   :: [Either Int String] -> String
addstrs'   xs = concat [s | Right s <- xs]
```

Logică propozițională

Propoziții

Dorim să definim în Haskell calculul propozițional clasic.

```
type Name = String
```

```
data Prop = Var Name  
         | F  
         | T  
         | Not Prop  
         | Prop :|: Prop  
         | Prop :&: Prop  
         deriving (Eq, Ord)
```

```
type Names = [Name]
```

```
type Env = [(Name, Bool)] -- evaluarea variabilelor
```

Afișarea unei propoziții

```

showProp :: Prop -> String
showProp (Var x)      = x
showProp F             = "F"
showProp T             = "T"
showProp (Not p)       = par ("~" ++ showProp p)
showProp (p :|: q)     = par (showProp p ++ "|" ++ showProp q)
showProp (p :&: q)     = par (showProp p ++ "&" ++ showProp q)

```

```

par :: String -> String
par s = "(" ++ s ++ ")"

```

```

instance Show Prop where
  show = showProp

```

Mulțimea variabilelor unei propoziții

```
prop :: Prop
prop = (Var "a" :&: Not (Var "b"))
```

```
> names prop
["a", "b"]
```

```
names    :: Prop -> Names
names (Var x)    = [x]
names F          = []
names T          = []
names (Not p)    = names p
names (p :|: q)   = nub (names p ++ names q)
names (p :&: q)   = nub (names p ++ names q)
```

```
Prelude> :m + Data.List
```

```
Prelude Data.List> nub [1,2,2,3,1,4,2]
[1,2,3,4]
```

-- elimina duplicatele

Evaluarea unei propoziții

```

type Env = [(Name,Bool)]  -- evaluarea variabilelor
eval    :: Env -> Prop -> Bool
lookUp  :: Eq a => [(a,b)] -> a -> b

```

```

lookUp env x = head [ y | (x',y) <- , x == x' ]
-- nu tratam cazurile de eroare

```

```

eval    e (Var x)           = lookUp e x
eval    e F                 = False
eval    e T                 = True
eval    e (Not p)           = not (eval e p)
eval    e (p |: q)          = eval e p || eval e q
eval    e (p :& q)           = eval e p && eval e q

```

Propoziții

Example

```
p0 :: Prop
p0 = (Var "a" :&: Not (Var "a"))
```

```
e0 :: Env
e0 = [( "a" , True )]
```

```
*Main> showProp p0
"(a&(~a))"
```

```
*Main> names p0
["a"]
```

```
*Main> eval e0 p0
False
```

```
*Main> lookUp e0 "a"
True
```


Cum funcționează evaluarea?

```

eval e0 (Var "a" :&: Not (Var "a"))
=
(eval e0 (Var "a")) && (eval e0 (Not (Var "a")))
=
(lookup e0 "a") && (eval e0 (Not (Var "a")))
=
True && (eval e0 (Not (Var "a")))
=
True && (not (eval e0 (Var "a")))
= ... =
True && False
=
False

```

Propoziții

Alte exemple

```
p1 :: Prop
p1 = (Var "a" :& Var "b") :|:
      (Not (Var "a") :& Not (Var "b"))
```

```
e1 :: Env
e1 = [("a", False), ("b", False)]
```

```
*Main> showProp p1
"((a&b)|((~a)&(~b)))"
```

```
*Main> names p1
["a", "b"]
```

```
*Main> eval e1 p1
True
```

```
*Main> lookUp e1 "a"
False
```

Generarea tuturor evaluărilor

```

envs :: Names -> [Env]
envs []          = [[]]
envs (x:xs)      = [ (x, False) : e | e <- envs xs ] ++
                   [ (x, True ) : e | e <- envs xs ]

```

Alternativă

```

envs :: Names -> [Env]
envs []          = [[]]
envs (x:xs)      = [ (x,b) : e | b <- bs, e <- envs xs ]
  where
    bs = [ False, True ]

```

Evaluări

```
envs [] = [[]]
```

```
envs ["b"]
= [("b", False) : []] ++ [("b", True) : []]
= [ [("b", False)], [("b", True) ]]
```

```
envs ["a", "b"]
= [("a", False) : e | e <- envs ["b"] ] ++
  [("a", True) : e | e <- envs ["b"] ]
= [("a", False) : [("b", False)], ("a", False) : [("b", True) ] ] ++
  [("a", True) : [("b", False)], ("a", True) : [("b", True) ] ]
= [ [ ("a", False), ("b", False) ], [ ("a", False), ("b", True) ],
    [ ("a", True), ("b", False) ], [ ("a", True), ("b", True) ] ]
```

Exercițiu: Scrieți o funcție care verifică dacă o propoziție este satisfiabilă.

```
satisfiable :: Prop -> Bool
```

Expresii

Expresii

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

```
showExp      :: Exp -> String
showExp (Lit n)      = show n
showExp (Add e1 e2) = par (showExp e1 ++ "+" ++ showExp
                           e2)
showExp (Mul e1 e2) = par (showExp e1 ++ "*" ++ showExp
                           e2)
```

```
par :: String -> String
par s = "(" ++ s ++ ")"
```

```
instance Show Exp where
  show = showExp
```

Expresii

```

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

```

Scrieți o funcție care evaluează expresiile:

```

> evalExp $ Add (Lit 2) (Mul (Lit 3) (Lit 3))
11

```

```

evalExp    :: Exp -> Int
evalExp    (Lit n)      = n
evalExp    (Add e1 e2)  = evalExp e1 + evalExp e2
evalExp    (Mul e1 e2)  = evalExp e1 * evalExp e2

```

Expresii

Exemple

```
ex0, ex1 :: Exp
ex0 = Add (Lit 2) (Mul (Lit 3) (Lit 3))
ex1 = Mul (Add (Lit 2) (Lit 3)) (Lit 3)
```

```
*Main> showExp ex0
"(2+(3*3))"
```

```
*Main> evalExp ex0
11
```

```
*Main> showExp ex1
"((2+3)*3)"
```

```
*Main> evalExp ex1
15
```


Expresii cu operatori

```
data    Exp    =    Lit Int
          |      Exp  :+:  Exp
          |      Exp  :*:  Exp
```

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

```
showExp :: Exp -> String
showExp (Lit n)    = show n
showExp (e :+: f) = par (showExp e ++ "+" ++ showExp f)
showExp (e :*: f) = par (showExp e ++ "*" ++ showExp f)
```

```
par :: String -> String
par s = "(" ++ s ++ ")"
```

Expresii cu operatori

Exemple

```
e0, e1 :: Exp
```

```
e0 = Lit 2 :+: (Lit 3 :*: Lit 3)
```

```
e1 = (Lit 2 :+: Lit 3) :*: Lit 3
```

```
*Main> showExp e0
```

```
"(2+(3*3))"
```

```
*Main> evalExp e0
```

```
11
```

```
*Main> showExp e1
```

```
"((2+3)*3)"
```

```
*Main> evalExp e1
```

```
15
```

Arbori

Arbori binari de căutare

```
data BinaryTree a =    Empty
                    | Node (BinaryTree a) a (BinaryTree a)
deriving Show
```

- Scrieți o funcție care determină înălțimea unui arbore.

```
height :: BinaryTree a -> Int
```

```
height Empty          = 0
```

```
height (Node l _ r) = 1 + max (height l) (height r)
```

- Scrieți o funcție care întoarce parcurgerea în inordine.

```
inord :: BinaryTree a -> [a]
```

```
inord Empty          = []
```

```
inord (Node l x r) = inord l ++ [x] ++ inord r
```

Arbori binari

```
data BTree a = Leaf a
              | Node (BTree a) (BTree a)
deriving Show
```

```
exTree = Node (Node (Leaf 'a') (Leaf 'b')) (Leaf 'c')
```

```
*Main> :t exTree
exTree :: BTree Char
```

```
*Main> exTree
Node (Node (Leaf 'a') (Leaf 'b')) (Leaf 'c')
```

Arbori binari

```
data BTree a = Leaf a
              | Node (BTree a) (BTree a)
```

```
showBT :: Show a => BTree a -> String
```

```
showBT (Leaf a)    = show a
```

```
showBT (Node t1 t2) = "(" ++ (showBT t1) ++ "),( "
                    ++ (showBT t2) ++ ")"
```

```
instance (Show a) => Show (BinaryTree a) where
    show = showBT
```

```
exTree = Node (Node (Leaf 'a') (Leaf 'b')) (Leaf 'c')
*Main> exTree
(( 'a' ),( 'b' )) ,( 'c' )
```

Clasa Foldable

Expresii

```
sumList :: [Int] -> Int  -- suma elementelor listei
sumList [] = 0
sumList (x:xs) = x + sumList xs
```

```
sumBTree :: BTree Int -> Int  -- suma elementelor arborelui
sumBTree (Leaf x) = x
sumBTree (Node x y) = (sumBTree x) + (sumBTree y)
```

Observăm ca se aplică aceeași metodă:

- se folosește definiția cu șabloane
- se definește funcția în cazul simplu (de oprire)
- se apelează funcția pe subtermeni, recursiv, și se agreghează rezultatele obținute.

Pentru liste putem folosi **foldr**:

```
sumList = foldr (+) 0
```


din nou **foldr**

foldr pe liste

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f i [] = i
foldr f i (x:xs) = f x (foldr f i xs)
```

Problema: să generalizăm **foldr** la alte structuri recursive.

Exemplu: arbori binari

```
data BTree a = Leaf a
              | Node (BTree a) (BTree a)
```

Cum definim "**foldr**" înlocuind listele cu date de tip **BTree** ?

"foldr" folosind BTree

```
data BTree a = Leaf a
              | Node (BTree a) (BTree a)
```

foldTree

```
foldTree :: (a -> b -> b) -> b -> BTree a -> b
```

```
foldTree f i (Leaf x) = f x i
```

```
foldTree f i (Node l r) = foldTree f (foldTree f i r) l
```

foldTree

```
data BTree a = Leaf a
              | Node (BTree a) (BTree a)
```

```
foldTree :: (a -> b -> b) -> b -> BTree a -> b
foldTree f i (Leaf x) = f x i
foldTree f i (Node l r) = foldTree f (foldTree f i r) l
```

```
myTree = Node (Node (Leaf 1)(Leaf 2))(Node (Leaf 3)(Leaf 4))
```

```
*Main> foldTree (+) 0 myTree
10
```

```
sumTree = foldTree (+) 0
```

clasa **Foldable**

<https://en.wikibooks.org/wiki/Haskell/Foldable>

<https://hackage.haskell.org/package/base-4.10.0.0/docs/Data-Foldable.html>

Data.Foldable

```
class Foldable t where  
    foldr    :: (a -> b -> b) -> b -> t a -> b
```

```
instance Foldable BTree where  
    foldr = foldTree
```

Observație: în definiția clasei **Foldable**, variabila de tip **t** nu reprezintă un tip concret ([a], Sum a) ci un **constructor de tip** (BTree)

clasa Foldable

Data.Foldable

```
class Foldable t where
    foldr    :: (a -> b -> b) -> b -> t a -> b
```

```
instance Foldable BTree where
    foldr = foldTree
```

```
tree1 = Node(Node(Leaf 1)(Leaf 2))(Node (Leaf 3)(Leaf 4))
treeS = Node (Node(Leaf "a")(Leaf "b"))
          (Node (Leaf "c")(Leaf "d"))
```

```
*Main> foldr (+) 0 tree1
10
```

```
*Main> foldr (++) [] treeS
"abcd"
```

clasa Foldable

Data.Foldable

```
class Foldable t where
    foldr    :: (a -> b -> b) -> b -> t a -> b
```

```
instance Foldable BTree where
    foldr = foldTree
```

```
tree1 = Node(Node(Leaf 1)(Leaf 2))(Node (Leaf 3)(Leaf 4))
treeS = Node (Node(Leaf "a")(Leaf "b"))
           (Node (Leaf "c")(Leaf "d"))
```

Avem definite automat și alte funcții precum: **foldl**, **foldr'**, **foldr1**,...

```
*Main> foldl (++) [] treeS
"1234"
*Main> foldl (+) 0 tree1
10
```

din nou **foldr**

Problema: să generalizăm **foldr** la alte structuri recursive.

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

Cum definim "**foldr**" înlocuind listele cu date de tip **Exp** ?

```
evalExp :: Exp -> Int
evalExp (Lit n)    = n
evalExp (Add e1 e2) = evalExp e1 + evalExp e2
evalExp (Mul e1 e2) = evalExp e1 * evalExp e2
```

Vrem să definim "**foldExp**" astfel încât

```
evalExp = foldExp fLit (+) (*)
```

din nou **foldr**

```

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

```

```

foldExp fLit fAdd fMul (Lit n)    = fLit n
foldExp fLit fAdd fMul (Add e1 e2) = fAdd v1 v2
                                where
                                v1 = foldExp fLit fAdd fMul e1
                                v2 = foldExp fLit fAdd fMul e2

```

```

foldExp fLit fAdd fMul (Mul e1 e2) = fMul v1 v2
                                where
                                v1 = foldExp fLit fAdd fMul e1
                                v2 = foldExp fLit fAdd fMul e2

```

```

evalExp = foldExp fLit (+) (*)
        where fLit (Lit x) = x

```


din nou **foldr**

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

```
foldExp fLit fAdd fMul (Lit n)      = fLit n
foldExp fLit fAdd fMul (Add e1 e2) = fAdd v1 v2
                                where ...
```

```
foldExp fLit fAdd fMul (Mul e1 e2) = fMul v1 v2
                                where ...
```

Ce tip are **foldExp**?

```
foldExp :: (Int -> b) -> (b -> b -> b) -> (b -> b -> b) -> Exp Int -> b
```

Pe săptămâna viitoare!