Programare funcțională

Tipuri de date structurate. Expresii.

Ioana Leuștean
Traian Florin Șerbănuță

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

1. Expresii aritmetice

2. Logică propozițională

Tipuri de date algebrice

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

Forma generală

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

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.

Expresii aritmetice

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
```

Logică propoziţională

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

Exemple

```
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
```

Pe săptămâna viitoare!