Programare declarativă[1]

Evaluarea expresiilor

Traian Florin Șerbănuță

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

11 noiembrie 2016

[1]bazat pe cursul Informatics 1: Functional Programming de la University of Edinburgh

Expresii

Expresii

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

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

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

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

Expresii

Exemple

```
e0, e1 :: Exp
e0 = Add (Lit 2) (Mul (Lit 3) (Lit 3))
e1 = Mul (Add (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
```

Expresii (forma infixată)

```haskell
data    Exp    =    Lit Int
               |    Exp `Add` Exp
               |    Exp `Mul` Exp
evalExp :: Exp -> Int
evalExp (Lit n)       = n
evalExp (e `Add` f) =    evalExp e + evalExp f
evalExp (e `Mul` f) =    evalExp e * evalExp f

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

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

Expresii (forma infixată)

Exemple

```
e0, e1 :: Exp
e0 = Lit 2 `Add` (Lit 3 `Mul` Lit 3)
e1 = (Lit 2 `Add` Lit 3) `Mul` Lit 3

*Main> showExp e0
"(2+(3*3))"

*Main> evalExp e0
11

*Main> showExp e1
"((2+3)*3)"

*Main> evalExp e1
15
```

Expresii cu operatori

```haskell
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 ca 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 propozițională

Propoziții

```haskell
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)]
```

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 ++ ")"
```

Mulțimea variabilelor unei propoziții

```haskell
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)
```

Evaluarea unei propoziții

Valuație

```haskell
eval    :: Env -> Prop -> Bool
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

lookUp :: Eq a => [(a,b)] -> a -> b
lookUp xys x = the [ y | (x',y) <- xys, x == x' ]
  where
  the [x] = x
```

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  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
    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 valuațiilor

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

Alternativă

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

Valuații

```
  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 )]]
```

Satisfiabilitate

```
satisfiable :: Prop -> Bool
satisfiable p = or [ eval e p | e <- envs (names p) ]
```

Satisfiabilitate

Exemplu

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

*Main> envs (names p1)
[[("a",False),("b",False)],
 [("a",False),("b",True)],
 [("a",True ),("b",False)],
 [("a",True ),("b",True )]]

*Main> [ eval e p1 | e <- envs (names p1) ]
[True,
 False,
 False,
 True]

*Main> satisfiable p1
True
```

Parțialitate

Tipul Opțiune

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

Argumente opționale

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

Rezultate opționale

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

Folosirea unui rezultat opțional

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

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

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

Variante

A sau B

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

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

```haskell
addints   :: [Either Int String] -> Int
addints   []                 = 0
addints   (Left n : xs)      = n + addints xs
addints   (Right s : xs) = addints xs
```

```haskell
addints' :: [Either Int String] -> Int
addints' xs = sum [n | Left n <- xs]
```

A sau B

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

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

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

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

Mini-Haskell

Sintaxă și Memorie

```
data   Hask   =   HTrue
                |   HFalse
                |   HIf  Hask  Hask  Hask
                |   HLit  Int
                |   HEq  Hask  Hask
                |   HAdd  Hask  Hask
                |   HVar  Name
                |   HLam  Name  Hask
                |   HApp  Hask  Hask

data   Value  =   VBool  Bool
                |   VInt  Int
                |   VList  [Value]
                |   VFun  (Value -> Value)

type   HEnv   =   [(Name, Value)]
```

Afișare și Egalitate pentru valori

```haskell
showValue :: Value -> String
showValue (VBool b)    = show b
showValue (VInt i)     = show i
showValue  (VList us) =
  "[" ++ concat (intersperse "," (map showValue us)) ++ "]"

eqValue :: Value -> Value -> Bool
eqValue (VBool b) (VBool c)    = b == c
eqValue (VInt i) (VInt j)      = i == j
eqValue (VList us) (VList vs) =
  and [ eqValue u v | (u,v) <- zip us vs ]
eqValue _ _                    = False
```

Observație

Funcțiile nu pot fi afișate nici testate dacă sunt egale.

Evaluarea expresiilor Mini-Haskell în Haskell

```
hEval :: Hask -> HEnv -> Value
hEval HTrue r          = VBool True
hEval HFalse r         = VBool False
hEval (HIf c d e) r    =
  hif (hEval c r) (hEval d r) (hEval e r)
  where hif (VBool b) v w = if b then v else w
hEval (HLit i) r       = VInt i
hEval (HEq d e) r      = heq (hEval d r) (hEval e r)
  where heq (VInt i) (VInt j) = VBool (i == j)
hEval (HAdd d e) r     = hadd (hEval d r) (hEval e r)
  where hadd (VInt i) (VInt j) = VInt (i + j)
hEval (HVar x) r       = lookUp r x
hEval (HLam x e) r     = VFun (\ v -> hEval e ((x,v):r))
hEval (HApp d e) r     = happ (hEval d r) (hEval e r)
  where happ (VFun f) v = f v

lookUp :: HEnv -> Name -> Value
lookUp x r = head [ v | (y,v) <- r, x == y ]
```

Evaluarea expresiilor Mini-Haskell în Haskell

Test

```
h0 =
 (HApp
   (HApp
     (HLam "x" (HLam "y" (HAdd (HVar "x") (HVar "y"))))
     (HLit 3))
   (HLit 4))

test_h0 = eqValue (hEval h0 []) (VInt 7)
```