

Programare declarativă¹

Monade Standard

Traian Florin Șerbănuță
Ioana Leuștean

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

¹bazat pe P. Wadler, The essence of functional programming și pe cursul Informatics I: Functional Programming de la University of Edinburgh

Evaluare cu efecte laterale

Lambda calcul cu întregi

Sintaxă

```
type Name = String
```

```
data Term = Var Name  
          | Con Integer  
          | Term :+: Term  
          | Lam Name Term  
          | App Term Term
```

Exemplu

λ -expresia $(\lambda x.x + x)(10 + 11)$

este definită astfel:

`pgm :: Term`

`pgm = App`

```
  (Lam "x" ((Var "x") :+: (Var "x")))
  ((Con 10) :+: (Con 11))
```

Valori

```
data Value = Num Integer  
            | Fun (Value  $\rightarrow$  M Value)  
            | Wrong
```

```
instance Show Value where  
  show (Num x) = show x  
  show (Fun _) = "<function>"  
  show Wrong   = "<wrong>"
```

Observații:

- Vom interpreta termenii în valori (M Value), unde M este o monadă; variind M se obțin comportamente diferite;
- Wrong reprezintă o eroare, de exemplu adunarea unor valori care nu sunt numere sau aplicarea unui termen care nu e funcție.

Medii de evaluare și evaluări

Interpretarea termenilor în monada M

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

```
interp :: Term -> Environment -> M Value
```

```
interp (Var x) env = lookupM x env
```

```
interp (Con i) _ = return $ Num i
```

```
interp (Lam x e) env = return $
```

```
  Fun $ \ v -> interp e ((x,v):env)
```

```
lookupM :: Name -> Environment -> M Value
```

```
lookupM x env = case lookup x env of
```

```
  Just v   -> return v
```

```
  Nothing -> return Wrong
```

Evaluare

Interpretarea adunării în monada M

```
interp (t1 :+: t2) env = do  
  v1 <- interp t1 env  
  v2 <- interp t2 env  
  add v1 v2
```

```
add :: Value -> Value -> M Value  
add (Num i) (Num j) = return (Num $ i + j)  
add _ _           = return Wrong
```

Evaluare

Interpretarea aplicării funcțiilor în monada M

```
interp (App t1 t2) env = do  
  f <- interp t1 env  
  v <- interp t2 env  
  apply f v
```

```
apply :: Value -> Value -> M Value  
apply (Fun k) v = k v  
apply _ _      = return Wrong
```


Testarea interpretorului

```
test :: Term -> String  
test t = showM $ interp t []
```

unde

```
showM :: Show a => M a -> String
```

este o funcție definită pentru fiecare monadă M în parte, i.e. pentru fiecare tip de efecte laterale dorit.

Interpretare în monada Identity

```
type M a = Identity a
```

```
showM :: Show a => M a -> String
```

```
showM = show . runIdentity
```

Unde monada Identity capturează transformarea identitate:

```
newtype Identity a = Identity { runIdentity :: a }
```

```
instance Monad Identity where
```

```
    return    = Identity
```

```
    m >=> k   = k (runIdentity m)
```

Observație:

```
runIdentity :: Identity a -> a
```

Obținem interpretorul standard discutat în cursurile trecute

Interpretare în monada Identitate

```
type M a = Identity a
```

```
showM :: Show a => M a -> String
```

```
showM = show . runIdentity
```

```
pgm :: Term
```

```
pgm = App
```

```
    (Lam "x" ((Var "x") :+: (Var "x")))
```

```
    ((Con 10) :+: (Con 11))
```

```
*Var0> test pgm
```

```
"42"
```

Interpretare în monada Optiune

Putem renunța la valoarea Wrong folosind monada **Maybe**

```
type M a = Maybe a
```

```
showM :: Show a => M a -> String
```

```
showM (Just a) = show a
```

```
showM Nothing = "<wrong>"
```

Putem acum înlocui rezultatele Wrong cu **Nothing**

```
lookupM x env = case lookup x env of
```

```
  Just v   -> return v
```

```
  Nothing -> Nothing
```

```
add (Num i) (Num j) = return (Num $ i + j)
```

```
add _ _ = Nothing
```

```
apply (Fun k) v = k v
```

```
apply _ _ = Nothing
```

Interpretare în monada Either String

Putem nuanța erorile folosind monada **Either String**

```
type M a = Either String a
showM (Left s)  = "Error:␣" ++ s
showM (Right a) = "Success:␣" ++ show a
```

Putem acum înlocui rezultatele Wrong cu valori **Left**

```
lookupM x env = case lookup x env of
  Just v  -> return v
  Nothing -> Left ("unbound␣variable␣" ++ x)
add (Num i) (Num j) = return $ Num $ i + j
add v1 v2
  = Left
    ("should␣be␣numbers:␣" ++ show v1 ++ ",␣" ++ show v2)
apply (Fun k) v = k v
apply v _
  = Left ("should␣be␣function:␣" ++ show v)
```

Interpretare în monada Either String

```
type M a = Either String a  
showM (Left s)  = "Error:␣" ++ s  
showM (Right a) = "Success:␣" ++ show a
```

```
pgm = App  
      (Lam "x" ((Var "x") :+: (Var "x"))) )  
      ((Con 10) :+: (Con 11))
```

```
*Var2> test pgm  
"Success: 42"
```

```
pgmE = App (Var "x") ((Con 10) :+: (Con 11))
```

```
*Var2> test pgmE  
"Error: unbound variable x"
```

Monada Stare

Control.Monad.State

```
newtype State s a = State { runState :: s -> (a, s) }
```

```
instance Monad (State s) where  
  return x = State (\ s -> (x,s))  
  m >>= k = State (\ s ->  
    let (x, s') = runState m s  
    in runState (k x) s')
```

```
get :: State s s                                -- produce starea curenta  
get = State (\ s -> (s,s))
```

```
put :: s -> State s ()                          -- schimba starea curenta  
put s = State (\ _ -> ((), s))
```

```
modify :: (s -> s) -> State s () -- modifica starea  
modify f = State (\ s -> ((), f s))
```

Interpretare în monada State

Adăugarea unui contor de instrucțiuni

```
data Term = ... | Count
```

```
type M a = State Integer a
```

```
showM ma = show a ++ "\n" ++ "Count:␣" ++ show s
           where (a, s) = runState ma 0
```

```
tickS = modify (+1)
```

```
add (Num i) (Num j) = tickS >> return (Num $ i + j)
add _ _             = return Wrong
```

```
apply (Fun k) v = tickS >> k v
apply _ _       = return Wrong
```

Iar evaluarea lui Count se face astfel:

```
interp Count _ = get >>= (\i -> return $ Num i)
```


Interpretare în monada State

```
data Term = ... | Count
```

```
type M a = State Integer a
```

```
showM ma = show a ++ "\n" ++ "Count:␣" ++ show s
           where (a, s) = runState ma 0
```

```
pgm :: Term
```

```
pgm = App
```

```
    (Lam "x" ((Var "x") :+: (Var "x")))
    ((Con 10) :+: (Con 11))
```

```
*Var3> test pgm
```

```
"42\nCount: 3"
```

Monada Writer

Control.Monad.Writer

Este folosită pentru a acumula (logging) informație produsă în timpul execuției

```
newtype Writer w a = Writer { runWriter :: (a, w) }
```

```
instance Monoid w => Monad (Writer w) where
```

```
  return x = Writer (x, mempty)
```

```
  ma >>= k = Writer $
```

```
    let (x, w)    = runWriter ma
```

```
        (x', w') = runWriter (k x)
```

```
    in (x', w 'mappend' w')
```

```
tell :: w -> Writer w ()
```

```
tell w = Writer ((), w)
```

Interpretare în monada Writer

Adăugarea unei instrucțiuni de afișare

```
data Term = ... | Out Term
```

```
type M a = Writer String a
```

```
showM :: Show a => M a -> String
```

```
showM ma = "Output:␣" ++ w ++ "\nValue:␣" ++ show a  
  where (a, w) = runWriter ma
```

```
interp (Out t) env = do  
  v <- interp t env  
  tell (show v ++ ";␣")  
  return v
```

- Out t se evaluează la valoarea lui t
- cu efectul lateral de a adăuga valoarea la șirul de ieșire.

Interpretare în monada Writer

```
data Term = ... | Out Term
```

```
type M a = Writer String a
```

```
showM :: Show a => M a -> String
```

```
showM ma = "Output:␣" ++ w ++ "\nValue:␣" ++ show a
  where (a, w) = runWriter ma
```

```
pgm :: Term
```

```
pgm = App
```

```
  (Lam "x" ((Var "x") :+: (Var "x")))
  ((Con 10) :+: (Con 11))
```

```
*Var4> test pgm
```

```
"Output: \nValue: 42"
```

Interpretare în monada listelor

Adăugarea unei instrucțiuni nedeterminate

```
data Term = ... | Amb Term Term | Fail
```

```
type M a = [a]
```

```
showM :: Show a => M a -> String
```

```
showM = show
```

```
interp Fail _ = []
```

```
interp (Amb t1 t2) env = interp t1 env ++ interp t2 env
```

```
*Var5> test (App (Lam "x" (Var "x" :+: Var "x")) (Amb (Con  
    1) (Con 2)))  
"[2,4]"
```

Monada Reader

Control.Monad.Reader

Face accesibilă o memorie imutabilă (environment)

```
newtype Reader r a = Reader { runReader :: r -> a }
```

```
instance Monad (Reader r) where
```

```
    return x = Reader (\ _ -> x)
```

```
    ma >=> k = Reader $ \ r ->
```

```
        let x = runReader ma r
```

```
        in runReader (k x) r
```

```
-- obține memoria
```

```
ask :: Reader r r
```

```
ask = Reader (\ r -> r)
```

```
-- modifică local memoria
```

```
local :: (r -> r) -> Reader r a -> Reader r a
```

```
local f ma = Reader $ \ r -> runReader ma (f r)
```

Interpretare în monada Reader

Eliminarea argumentului Environment – expresii de bază și lookup

```
type M a = Reader Environment a
```

```
showM :: Show a => M a -> String
```

```
showM ma = show $ runReader ma []
```

```
interp :: Term -> M Value
```

```
interp (Var x) = lookupM x
```

```
interp (Con i) = return $ Num i
```

```
lookupM :: Name -> M Value
```

```
lookupM x = do
```

```
  env <- ask
```

```
  case lookup x env of
```

```
    Just v   -> return v
```

```
    Nothing -> return Wrong
```

Interpretare în monada Reader

Eliminarea argumentului Environment – operatori binari și funcții

```
interp (t1 :+: t2) = do
  v1 <- interp t1
  v2 <- interp t2
  add v1 v2

interp (App t1 t2) = do
  f <- interp t1
  v <- interp t2
  apply f v

interp (Lam x e) = do
  env <- ask
  return $ Fun $ \ v ->
    local (const ((x,v):env)) (interp e)
```


Interpretare în monada Reader

```
type M a = Reader Environment a
```

```
showM :: Show a => M a -> String
```

```
showM ma = show $ runReader ma []
```

```
interp :: Term -> M Value
```

```
pgm :: Term
```

```
pgm = App
```

```
  (Lam "x" ((Var "x") :+: (Var "x")))
```

```
  ((Con 10) :+: (Con 11))
```

```
*Var6> test pgm
```

```
"42"
```