Programare declarativă

Evaluare cu efecte laterale

În acest curs:

- vom defini un mini-limbaj asemănător cu limbajul Mini Haskell definit în cursurile trecute
- vom defini semantica limbajului folosind o monadă generică M
- înlocuind M cu monadele standard studiate anterior vom obține variații ale semanticii generale, care vor fi particularizate prin tipul de efecte surprins de fiecare monadă

Sintaxă abstractă

Lambda calcul cu întregi Sintaxă

```
type Name = String

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

Program - 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))
```

Program - Exemplu

```
pgm :: Term
pgm = App
   (Lam "y"
      (App
         (App
            (Lam "f"
               (Lam "y"
                  (App (Var "f") (Var "y"))
               )
            )
            (Lam "x"
               (Var "x" :+: Var "y")
            )
         )
         (Con 3)
      )
   )
   (Con 4)
```

Valori și medii de evaluare

```
data Value = Num Integer
           | Fun (Value -> 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 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.

Evaluare - variabile și valori

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

Interpretarea termenilor în monada ‘M‘

```
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 - adunare

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

Interpretarea adunării în monada ‘M‘

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

Evaluare - aplicarea funcțiilor

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

Interpretarea aplicării funcțiilor în monada 'M'

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

-- k :: Value -> M Value
```

Testarea interpretorului

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

unde

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

este o funcție definită special pentru fiecare tip de efecte laterale dorit.

Testarea interpretorului

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

unde

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

este o funcție definită special pentru fiecare tip de efecte laterale dorit.

Exemplu de program

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

test pgmW  -- apelul pentru testare
```

Interpretor monadic

```
data Value = Num Integer
           | Fun (Value -> M Value)
           | Wrong

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

În continuare vom înlocui monada M cu:

- Identity
- **Maybe**
- **Either String**
- monada listelor
- Writer
- Reader
- State

Interpretare în monada ‘Identity‘

Monada ‘Identity‘ este "efectul identitate".

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

instance Monad Identity where
    return a              = Identity a
    ma >>= k = k (runIndentity ma)
```

Interpretare în monada ‘Identity‘

Monada ‘Identity‘ este "efectul identitate".

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

instance Monad Identity where
    return a              = Identity a
    ma >>= k = k (runIndentity ma)
```

Pentru a particulariza interpretorul definim

```
type M a = Identity a

showM :: Show a => M a -> String
showM = show . runIdentity
```

Interpretare în monada ‘Identity‘

Monada ‘Identity‘ este "efectul identitate".

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

instance Monad Identity where
    return a              = Identity a
    ma >>= k = k (runIndentity ma)
```

Pentru a particulariza interpretorul definim

```
type M a = Identity a

showM :: Show a => M a -> String
showM = show . runIdentity
```

Obținem interpretorul standard, asemănător celui discutat pentru limbajul Mini-Haskell.

Interpretare folosind monada ‘Identity‘

```
type M a = Identity a

showM :: Show a => M a -> String
showM = show . runIdentity

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

*Var0> test pgm
"42"
```

Interpretare în monada ‘Maybe‘ (opțiune)

```
data Maybe a = Nothing | Just a

instance Monad Maybe where
  return = Just
  Just a  >>= k  = k a
  Nothing >>= _  = Nothing
```

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>"
```

Interpretare în monada ‘Maybe‘

Putem acum înlocui rezultatele ‘Wrong‘ cu ‘Nothing‘

```
type M a = Maybe a

lookupM :: Name -> Environment -> M Value
lookupM x env = case lookup x env of
  Just v  -> return v
  Nothing -> Nothing

add :: Value -> Value -> M Value
add (Num i) (Num j) = return (Num $ i + j)
add _ _             = Nothing

apply :: Value -> Value -> M Value
apply (Fun k) v = k v
apply _ _       = Nothing
```

Interpretare în monada ‘Either String‘

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

instance Monad (Either err) where
  return = Right
  Right a >>= k  = k a
  err     >>= _  = err
```

Putem nuanța erorile folosind monada ‘Either String‘

```
type M a = Either String a

showM :: Show a => M a -> String
showM (Left s)  = "Error: " ++ s
showM (Right a) = "Success: " ++ show a
```

Interpretare în monada ‘Either String‘

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

```
type M a = Either String a

lookupM :: Name -> Environment -> M Value
lookupM x env = case lookup x env of
  Just v  -> return v
  Nothing -> Left ("unbound variable " ++ x)

add :: Value -> Value -> M Value
add (Num i) (Num j) = return $ Num $ i + j
add v1 v2           = Left $
  "Expected numbers: " ++ show v1 ++ ", " ++ show v2

apply :: Value -> Value -> M Value
apply (Fun k) v = k v
apply v _       = Left $
  "Expected function: " ++ show v
```

Interpretare în monada ‘Either String‘

```
type M a = Either String a

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

Interpretare în monada ‘Either String‘

```
type M a = Either String a

showM :: Show a => M a -> String
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 listelor (a funcțiilor nedeterministe)

```
instance Monad [] where
  return a = [a]
  ma >>= k = [b | a <- ma, b <- k a]
```

Rezultatul funcției e lista tuturor valorilor posibile.

```
> [4,9,25] >>= \x -> [(sqrt x), -(sqrt x)]
[2.0,-2.0,3.0,-3.0,5.0,-5.0]
```

Interpretare în monada listelor

Adăugarea unei instrucțiuni nedeterministe

```
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

pgm = (App (Lam "x" (Var "x" :+: Var "x"))
       (Amb (Con 1) (Con2)))
> test pgm
"[2,4]"
```

Monada ‘Writer‘

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

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

instance Monoid log => Monad (Writer log) where
  return a = Writer (a, mempty)
  ma >>= k = let (a, log1) = runWriter ma
                 (b, log2) = runWriter (k a)
             in Writer (b, log1 `mappend` log2)
```

Funcție ajutătoare

```
tell :: log -> Writer log ()
tell log = Writer ((), log)  -- produce mesajul
```

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 ++ "  Value: " ++ 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 ++ "  Value: " ++ show a
  where (a, w) = runWriter ma

pgmW = App
          (Lam "x" ((Var "x") :+: (Var "x")))
          ((Out (Con 10)) :+: (Out (Con 11)))

> test pgm
"Output: 10; 11; Value: 42"
```

Monada ‘Reader‘

Face accesibilă o memorie (environment) nemodificabilă (imuabilă)

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

instance Monad (Reader env) where
  return = Reader const
  ma >>= k = Reader f
    where f env = let a = runReader ma env
                  in  runReader (k a) env
```

Monada ‘Reader‘

Face accesibilă o memorie (environment) nemodificabilă (imuabilă)

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

instance Monad (Reader env) where
  return = Reader const
  ma >>= k = Reader f
    where f env = let a = runReader ma env
                  in  runReader (k a) env
```

Funcții ajutătoare

```
ask :: Reader r r    -- obtine memoria
ask = Reader id  -- Reader (\r -> r)

-- modifica 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‘

```
type Environment = [(Name, Value)]
type M a = Reader Environment a

showM :: Show a => M a -> String
showM ma = show $ runReader ma []
```

Funcția de interpretare era definită astfel:

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

Deoarece interpretăm în monada ‘Reader Enviroment a‘ signatura funcției de interpretare este:

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

Interpretare în monada ‘Reader‘

Interpretarea expresiilor de bază și căutare(‘lookup‘)

```
type Environment = [(Name, Value)]
type M a = Reader Environment a

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‘

```
type Environment = [(Name, Value)]
interp :: Term -> M Value
```

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 Environment = [(Name, Value)]
type M a = Reader Environment a

showM :: Show a => M a -> String
showM ma = show $ runReader ma []

interp :: Term ->  M Value

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

> test pgm
"42"
```

Monada ‘State‘

```
newtype State state a =
  State { runState :: state -> (a, state) }
instance Monad (State state) where
  return a = State (\ s -> (a, s))
  ma >>= k = State (\state ->
      let (a, aState) = runState ma state
       in runState (k a) aState)
```

Funcții ajutătoare

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

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

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

Interpretare în monada ‘State‘

Adăugăm un contor de instrucțiuni ‘Count‘, valoarea acestui contor reprezentând starea.
Astfel variabilele care reprezintă starea sunt numere întregi.

```
data Term = ... | Count

type M a = State Integer a

showM :: Show a => M a -> String
showM ma = show a ++ "\n" ++ "Count: " ++ show s
           where (a, s) = runState ma 0

interp Count _ = do
                   i <- get
                   return (Num i)
```

Interpretare în monada ‘State‘

Creștem starea (contorul) la fiecare instrucțiune

```
tickS :: M ()
tickS = modify (+1)  -- \s ->((), (s+1))

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

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

Interpretare în monada ‘State‘

```
data Term = ... | Count

type M a = State Integer a

showM :: Show a => M a -> String
showM ma = show a ++ "\n" ++ "Count: " ++ show s
           where (a, s) = runState ma 0

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

> test pgm
"42\nCount: 3"
```

Pe săptămâna viitoare!