Programare funcțională

Introducere în programarea funcțională folosind Haskell
C11- Seriile 23 si 25

Ana Iova
Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Monade

Clasa de tipuri Monad

```
class Applicative m => Monad m where
    (>>=)  :: m a -> (a -> m b) -> m b
    (>>)   :: m a -> m b -> m b
    return :: a -> m a
```

În Haskell, monada este o clasă de tipuri!

Clasa **Monad** este o extensie a clasei Applicative!

- m a este tipul comenzilor care produc rezultate de tip a (și au efecte laterale)
- a -> m b este tipul continuărilor / a funcțiilor cu efecte laterale
- >>= este operația de „secvențiere” a comenzilor

Notația do pentru monade

```
(>>=)  :: m a -> (a -> m b) -> m b
(>>)   :: m a -> m b -> m b
```

Notația cu operatori	Notația **do**
e >>= \x -> rest	x <- e rest
e >>= _ -> rest	e rest
e >> rest	e rest

```
binding' :: IO ()
binding' =
  getLine >>= putStrLn

binding :: IO ()
binding = do
  name <- getLine
  putStrLn name
```

Notația do pentru monade

```
twoBinds' :: IO ()
twoBinds' =
      putStrLn "name pls: " >>
      getLine >>=

      \name ->
       putStrLn "age pls :" >>
       getLine >>=

      \age ->
       putStrLn ("hello:  "
                 ++ name ++ " who is: "
                 ++ age ++ " years old.")
```

Notația do pentru monade

```
twoBinds :: IO ()
twoBinds = do
    putStrLn "name pls: "
    name <- getLine

    putStrLn "age pls: "
    age <- getLine

    putStrLn ("hello: "
              ++ name ++ " who is: "
              ++ age ++ " years old.")
```

Notația do pentru monade

```
(>>=)  :: m a -> (a -> m b) -> m b
(>>)   :: m a -> m b -> m b
```

Notația cu operatori	Notația **do**
e >>= \x -> rest	x <- e rest
e >>= _ -> rest	e rest
e >> rest	e rest

De exemplu

```
e1 >>= \x1 -> e2 >> e3
```

devine

Notația do pentru monade

```
(>>=)  :: m a -> (a -> m b) -> m b
(>>)   :: m a -> m b -> m b
```

Notația cu operatori	Notația **do**
e >>= \x -> rest	x <- e rest
e >>= _ -> rest	e rest
e >> rest	e rest

De exemplu

```
e1 >>= \x1 -> e2 >> e3
```

devine

```
do
  x1 <- e1
  e2
  e3
```

Notația do pentru monade

```
(>>=)  :: m a -> (a -> m b) -> m b
(>>)   :: m a -> m b -> m b
```

De exemplu

```
e1 >>= \x1 -> e2 >>= \x2 -> e3 >>= \_ -> e4 >>= \x4 -> e5
```

devine

Notația do pentru monade

```
(>>=)  :: m a -> (a -> m b) -> m b
(>>)   :: m a -> m b -> m b
```

De exemplu

```
e1 >>= \x1 -> e2 >>= \x2 -> e3 >>= \_ -> e4 >>= \x4 -> e5
```

devine

```
do
  x1 <- e1
  x2 <- e2
  e3
  x4 <- e4
  e5
```

Functor și Applicative definiți cu return și >>=

```
instance Monad M where
  return a = ...
  ma >>= k = ...

instance Applicative M where
  pure = return
  mf <*> ma = do
       f <- mf
       a <- ma
            return (f a)
  -- mf >>= (\f -> ma >>= (\a -> return (f a)))

instance Functor M where
  fmap f ma = pure f <*> ma
  --  ma >>= \a -> return (f a)
  --  ma >>= (return . f)
```

Exemple de efecte laterale

I/O	Monada **IO**
Parțialitate	Monada **Maybe**
Excepții	Monada **Either**
Nedeterminism	Monada [] (listă)
Logging	Monada Writer
Stare	Monada State
Memorie read-only	Monada Reader

Monada listă

Monada pentru liste – nedeterminism

O computatie care intoarce un rezultat nedeterminist nu este o functie pura, dar poate fi transformata intr-o functie pura transformand rezultatul sau din tipul a in tipul [a].

In esenta, construim o functie care returneaza toate rezultatele posibile in acelasi timp.

Monada pentru liste – nedeterminism

Exemplu:

- Un joc de sah care evalueaza mutarile viitoare
- Trebuie sa anticipeze mutarile unui adversar nedeterminist
- Doar o mutare a adversarului se va concretiza, dar toate mutarile posibile trebuie luate in calcul in planificarea unei strategii
- O operatie de baza intr-un astfel de program este `move`: ia o stare a tablei de sah si returneaza o noua stare a tablei dupa ce o miscare a fost efectuata
- Dar ce ar trebui sa returneze `move`? Sunt multe mutari posibile in fiecare situatie, rezultatul fiind nedeterminist
- Ca sa putem compune astfel de mutari, trebuie sa definim o monada.

Instanta de Monade pentru liste

Functiile din clasa **Monad** specializate pentru liste:

```
(>>=)  :: [a] -> (a -> [b]) -> [b]
return :: a -> [a]
```

```
instance Monad [] where
  return x = [x]
  xs >>= f = concat (map f xs) -- [vb | va <- ma, vb
      <- k va]
```

Monada pentru liste – exemplu

```
twiceWhenEven :: [Integer] -> [Integer]
twiceWhenEven xs = do
    x <- xs
    if even x
       then [x*x, x*x]
       else [x*x]
```

Monada pentru liste – exemplu

```
radical :: Float -> [Float]
radical x
      | x >= 0 = [negate (sqrt x), sqrt x]
      | x < 0  = []

solEq2 :: Float -> Float -> Float -> [Float]
solEq2 0 0 c = []          -- a * x^2 + b * x + c = 0
solEq2 0 b c = return ((negate c) / b)
solEq2 a b c = do
  rDelta <- radical (b * b - 4 * a * c)
  return (negate b + rDelta) / (2 * a)
```

Monada Maybe(a funcțiilor parțiale)

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

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

Monada Maybe – exemplu

```
radical :: Float -> Maybe Float
radical x
    | x >= 0 = return (sqrt x)
    | x < 0  = Nothing

solEq2 :: Float -> Float -> Float -> Maybe Float
solEq2 0 0 0 = return 0     -- a * x^2 + b * x + c = 0
solEq2 0 0 c = Nothing
solEq2 0 b c = return ((negate c) / b)
solEq2 a b c = do
  rDelta <- radical (b * b - 4 * a * c)
  return ((negate b + rDelta) / (2 * a))
```

Monada Either(a excepțiilor)

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

instance Monad (Either err) where
  return = Right
  Right va >>= k  = k va
  err      >>= _  = err
  -- Left verr >>=_ = Left verr
```

Monada Either – exemplu

```
radical :: Float -> Either String Float
radical x
    | x >= 0 = return (sqrt x)
    | x < 0  = Left "radical: argument negativ"

solEq2 :: Float -> Float -> Float -> Either String
    Float
solEq2 0 0 0 = return 0     -- a * x^2 + b * x + c = 0
solEq2 0 0 c = Left "Nu are solutii"
solEq2 0 b c = return ((negate c) / b)
solEq2 a b c = do
    rDelta <- radical (b * b - 4 * a * c)
    return ((negate b + rDelta) / (2 * a))
```

Monada Writer (variantă simplificată)

```
newtype Writer log a = Writer {runWriter :: (a, log)}
  -- a este parametru de tip

instance Monad (Writer String) where
  return va =  Writer (va, "")
  ma >>= k =
      let (va, log1) = runWriter ma
          (vb, log2) = runWriter (k va)
        in Writer (vb, log1 ++ log2)
```

Monada Writer - Exemplu logging

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

tell :: log -> Writer log ()
tell msg = Writer ((), msg)

logIncrement :: Int -> Writer String Int
logIncrement x = do
    tell ("increment: " ++ show x ++ "\n")
    return (x + 1)
logIncrement2 :: Int -> Writer String Int
logIncrement2 x = do
    y <- logIncrement x
  logIncrement y

Main> runWriter (logIncrement2 13)
(15,"increment: 13\nincrement: 14\n")
```

Vacanta placuta!