Programare funcțională

Introducere în programarea funcțională folosind Haskell

C10

Claudia Chiriță
Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Monade

Clase de tipuri

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

class Functor m => Applicative m where
   pure :: a -> m a
  (<*>) :: m (a -> b) -> m a -> m b

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

   ma >> mb = ma >>= \_ -> mb
```

Proprietățile monadelor

Asociativitate și element neutru

Operația >>= este asociativă și are element neutru **return**

- Element neutru (la dreapta):

```
(return x) >>= g = g x
```

- Element neutru (la stânga):

```
x >>= return = x
```

- Associativitate:

```
(f >>= g) >>= h = f >>= (\ x -> (g x >>= h))
```

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
```

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 ("y helo thar: "
                 ++ 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 ("y helo thar: "
              ++ name ++ " who is: "
              ++ age ++ " years old.")
```

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 Maybe (a funcțiilor parțiale)

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

Funcțiile din clasa **Monad** specializate pentru **Maybe**:

```
(>>=) :: Maybe a -> (a -> Maybe b) ->  Maybe b
return :: a -> Maybe a

instance Monad Maybe where
  return = Just

  Just va >>= f   = f va
  Nothing >>= _   = Nothing
```

Monada Maybe – exemplu

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

-- a * x^2 + b * x + c = 0
solEq2 :: Float -> Float -> Float -> Maybe Float
solEq2 0 0 0 = return 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 Listă (nedeterminism)

O computație care întoarce un rezultat nedeterminist nu este o funcție pură.

Poate fi transformată într-o funcție pură transformând rezultatul său din tipul a în tipul [a].

În esență, construim o funcție care returnează toate rezultatele posibile în același timp.

Monada Listă (nedeterminism)

Funcțiile 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)

concat :: [[a]] -> [a]
```

Monada Listă – exemplu

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

Monadele Maybe și listă – exemplu

```
-- a * x^2 + b * x + c = 0
solEq2All :: Float -> Float -> Float -> Maybe [Float]
solEq2All 0 0 0 = return [0]
solEq2All 0 0 c = Nothing
solEq2All 0 b c = return [negate c / b]
solEq2All a b c = do
        rDelta <- radical (b * b - 4 * a * c)
        let s1 = (negate b + rDelta) / (2 * a)
        let s2 = (negate b - rDelta) / (2 * a)
        return [s1,s2]
```

Monada Either (a excepțiilor)

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

Funcțiile din clasa **Monad** specializate pentru **Either**:

```
(>>=) :: Either err a -> (a -> Either err b) ->
                Either err b
return :: a -> Either err a

instance Monad (Either err) where
        return = Right

        Right va >>= f = f va
             err >>= _ = err
  -- Left verr >>= _ = Left verr
```

Monada Either – exemplu

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

-- a * x^2 + b * x + c = 0
solEq2AllEither :: Float -> Float -> Float -> Either
    String [Float]
solEq2AllEither 0 0 0 = return [0]
solEq2AllEither 0 0 c = Left "ecuatie: fara solutie"
solEq2AllEither 0 b c = return [negate c / b]
solEq2AllEither a b c = do
    rDelta <- radicalEither (b * b - 4 * a * c)
    let s1 = (negate b + rDelta) / (2 * a)
    let s2 = (negate b - rDelta) / (2 * a)
    return [s1,s2]
```

Monada Writer (variantă simplificată)

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

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

Funcțiile din clasa **Monad** specializate pentru Writer:

```
(>>=) :: Writer log a -> (a -> Writer log b) ->
            Writer log b
return :: a -> Writer log a

instance Monad (Writer String) where
  return va =  Writer (va, "")
  ma >>= f =   let (va, log1) = runWriter ma
                   (vb, log2) = runWriter (f 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 ++ "--")
    return (x + 1)

logIncrement2 :: Int -> Writer String Int
logIncrement2 x = do
    y <- logIncrement x
    logIncrement y

*C10> runWriter (logIncrement2 13)
(15,"increment:13--increment:14")
```

Monada Reader (stare nemodificabilă)

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

ask :: Reader env env
ask = Reader id

instance Monad (Reader env) where
  return = Reader const
  -- return x = Reader (\_ -> x)

  ma >>= k = Reader f
    where
    f env = let va = runReader ma env
      in runReader (k va) env
```

Monada Reader - exemplu

```
tom :: Reader String String
tom = do
  env <- ask -- gives the environment (here a String)
  return (env ++ " This is Tom.")

jerry :: Reader String String
jerry = do
  env <- ask
  return (env ++ " This is Jerry.")

tomAndJerry :: Reader String String
tomAndJerry = do
    t <- tom
    j <- jerry
    return (t ++ "\n" ++ j)

runJerryRun :: String
runJerryRun = runReader tomAndJerry "Who is this?"
```

Pe data viitoare!