Programare declarativă — Efecte laterale

Traian Șerbănuță (33) Ioana Leuștean (34)

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

Efecte laterale

Logging în C

```
#include <iostream>
#include <sstream>
using namespace std;
ostringstream log;
int increment(int x) {
  log << "Called increment with argument " << x << endl;
  return x + 1;
}

int main() {
  int x = increment(increment(2));
  cout << "Result: " << x << endl << "Log: " << endl << log.str();
}
```

Fiecare apel al lui `increment` produce un mesaj. Mesajele se acumulează.

Stare în C

```
#include <iostream>
using namespace std;
const uint32_t MULTIPLIER = 1664525;
const uint32_t INCREMENT = 1013904223;
uint32_t seed = 0;

uint32_t rnd() {
  seed = MULTIPLIER * seed + INCREMENT; return seed;
}

void randomize(uint32_t x) { seed = x; }

int main() {
  randomize(1237853);
  cout << "Random1: " << rnd() << " Random2: " << rnd() << endl;
}
```

Fiecare apel al lui `rnd` citește starea existentă și o modifică.

Logging în Haskell

„Îmbogățim" rezultatul funcției cu mesajul de log.

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

Observații

- `newtype` seamana cu `data`, se foloseste cand avem un singur constructor si un singur tip de date
- datele de tip `Writer log a` sunt definite folosind *înregistrări*
- o dată de tip `Writer log a` are una din formele

  `` `Writer (va,vlog)` *sau* `Writer {runWriter = (va,vlog)}` ``

unde `va :: a` și `vlog :: log`

- `runWriter` este funcția proiecție:

  `runWriter :: Writer log a -> (a, log)`

  de exemplu `runWriter (Writer (1,"msg")) = (1,"msg")`

Logging în Haskell

„Îmbogățim” rezultatul funcției cu mesajul de log.

```haskell
newtype Writer log a = Writer { runWriter :: (a, log) }
-- newtype seamana cu data, se foloseste cand avem un singur
-- constructor  si un singur tip de date

logIncrement :: Int -> Writer String Int
logIncrement x = Writer
    (x + 1, "Called increment with argument " ++ show x ++ "\n")
```

Problemă: Cum calculăm `logIncrement (logIncrement x)`?

Logging în Haskell

„Îmbogățim” rezultatul funcției cu mesajul de log.

```
newtype Writer log a = Writer { runWriter :: (a, log) }
-- newtype seamana cu data, se foloseste cand avem un singur
-- constructor  si un singur tip de date

logIncrement :: Int -> Writer String Int
logIncrement x = Writer
    (x + 1, "Called increment with argument " ++ show x ++ "\n")
```

Problemă: Cum calculăm `logIncrement (logIncrement x)`?

```
logIncrement2 :: Int -> Writer String Int
logIncrement2 x = let (y, log1) = runWriter (logIncrement x)
                      (z, log2) = runWriter (logIncrement y)
                  in  Writer (z, log1 ++ log2)
```

Se poate! . . . dar nu vrem să facem asta pentru fiecare funcție.

Stare în Haskell

Rezultatul este acum o funcție, care dată fiind starea dinaintea execuției, produce un rezultat (folosind eventual starea) și starea cea nouă.

```
newtype State state a = State { runState :: state -> (a, state) }
rnd :: State Word32 Word32
rnd = State f
  where f seed = (seed', seed')
        seed' = cMULTIPLIER * seed + cINCREMENT
```

Problemă: Cum calculăm `rnd() + rnd()`?

Stare în Haskell

Rezultatul este acum o funcție, care dată fiind starea dinaintea execuției, produce un rezultat (folosind eventual starea) și starea cea nouă.

```
newtype State state a = State { runState :: state -> (a, state) }
rnd :: State Word32 Word32
rnd = State f
  where f seed = (seed', seed')
        seed' = cMULTIPLIER * seed + cINCREMENT
```

Problemă: Cum calculăm `rnd() + rnd()`?

```
rnd2 :: State Word32 Word32
rnd2 = State f
  where f seed = let (rnd1, seed1) = runState rnd seed
                     (rnd2, seed2) = runState rnd seed1
                 in  (rnd1 + rnd2, seed2)
```

Se poate! . . . dar nu vrem să facem asta pentru fiecare funcție.

Exemple de efecte laterale

- I/O
- Logging
- Stare
- Excepții
- Partialitate
- Nedeterminism
- Memorie read-only

Exemple de efecte laterale

- I/O
- Logging
- Stare
- Excepții
- Partialitate
- Nedeterminism
- Memorie read-only

N-ar fi frumos ca, pentru orice computație cu efecte laterale, să putem folosi notația do, ca pentru I/O ?

```
read2 = do
  x <- readLn :: IO Int
  y <- readLn :: IO Int
  return x + y
```

Exemplele precedente în notație do

Logging

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

RND

```
rnd2 :: State Word32 Word32
rnd2 = do
  r1 <- rnd
  r2 <- rnd
  return (r1 + r2)
```

Monade

Cum compunem funcții cu efecte laterale

Problema generală

Dată fiind funcția `f :: a -> m b` și funcția `g :: b -> m c`, vreau să obțin o funcție `g # f :: a -> m c` care este „compunerea” lui `g` și `f`, propagând efectele laterale.

Soluție

Transform `g :: b -> m c` în `bind g :: m b -> m c` pentru un `bind` „cu proprietăți bune”

```
bind :: (b -> m c) -> m b -> m c
(#) :: (b -> m c) -> (a -> m b) -> (a -> m c)
g # f = bind g . f
```

Observație

`bind` și `(#)` sunt `(=<<)` și `(<=<)` din Control.Monad

Clasa de tipuri Monad

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

- `m a` — tipul **computațiilor** care produc rezultate de tip a (și au efecte laterale)
- Tipul `a -> m b` este tipul continuărilor / a funcțiilor cu efecte laterale
- `>>=` este operația de „secvențiere” a computațiilor
- `bind k ma = ma >>= k`, adică `bind = flip (>>=)`
- `(g # f) x = (bind g . f) x = bind g (f x) = f x >>= g`

Applicative va fi discutată mai încolo

Notația do pentru monade

- `x <- e; rest` devine `e >>= \x -> rest`
- `e; rest` devine `e >> \_ -> rest`

De exemplu

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

devine

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

Proprietățile monadelor

Pe scurt

- Operația de compunere # a continuărilor
  - este asociativă
  - are element neutru `return`
- Asigură consistență notației do

Pe mai puțin scurt

- Element neutru (la dreapta): `return # g = g`
  `(return x) >>= g = g x`
- Element neutru (la stânga): `g # return = g`
  `x >>= return = x`
- Associativitate: `(f # g) # h = f # (g # h)`
  `(fm >>= g) >>= h = fm >>= \ x -> (g x >>= h)`

Monade standard

Exemple de efecte laterale

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

Monada Writer (variantă simplificată)

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

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

Monada Writer (variantă simplificată)

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

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

instance Monad (Writer String) where
  return a = Writer (a, "")     -- a este variabila
  ma >>= k = let (a, log1) = runWriter ma
                 (b, log2) = runWriter (k a)
             in Writer (b, log1 ++ log2)
```

Observație

În definițiile de mai sus am folosit a pentru a desemna un tip, dar și o variabilă de acel tip, contextul în care apare clarifică modul de utilizare. Convenții similare: `b` este o variabilă de tip `b`, `log1` si `log2` sunt variabile de tip `log`, iar `ma` este o variabilă de tip monadic `(m a)`.

Monada Writer — Exemplu logging

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

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

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")
```

Monada State

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

Dacă `ma :: State state a` atunci

`ma = State f` *sau* `ma = State {runState =f}`

unde `f :: state -> (a, state)`

Monada State

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

Dacă `ma :: State state a` atunci

`ma = State f` *sau* `ma = State {runState =f}`

unde `f :: state -> (a, state)`

```
instance Monad (State state) where
  return a = State (\s -> (a, s))
-- return a = State f where f = \s -> (a,s)

  ma >>= k = State g
    where g state = let (a, aState) = runState ma state
                    in runState (k a) aState

-- ma :: State state a , runState ma :: state -> (a, state)
-- k :: a -> State state b,  runState (k a) :: state -> (b, state)
-- ma >>= k :: State state b
--  g :: state -> (b, state)
```

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 g
    where g state = let (a, aState) = runState ma state
                    in runState (k a) aState
```

Funcții ajutătoare:

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

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

Monada State — exemplu random

```
newtype State state a = State {runState :: state -> (a,state)}
get :: State state state
get = State (\s -> (s,s))
modify :: (state -> state) -> State state ()
modify f = State (\s -> ((), f s))
```

Monada State — exemplu random

```
newtype State state a = State {runState :: state -> (a,state)}
get :: State state state
get = State (\s -> (s,s))
modify :: (state -> state) -> State state ()
modify f = State (\s -> ((), f s))

cMULTIPLIER, cINCREMENT :: Word32
cMULTIPLIER = 1664525 ; cINCREMENT = 1013904223

rnd, rnd2 :: State Word32 Word32
rnd = do modify (\seed -> cMULTIPLIER * seed + cINCREMENT)
         get
rnd2 = do r1 <- rnd
          r2 <- rnd
          return (r1 + r2)

Main> runState rnd2 0
(2210339985,1196435762)
```

Monada `Either` (a excepțiilor)

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

Monada Either (a excepțiilor)

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

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

Monada Either (a excepțiilor)

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

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

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 let delta = b * b - 4 * a * c
                  rDelta <- radical delta
                  return (negate b + rDelta) / (2 * a)
```

Monada Maybe (a funcțiilor parțiale)

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

Monada Maybe (a funcțiilor parțiale)

```
data Maybe a = Nothing | Just a

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

Monada Maybe (a funcțiilor parțiale)

```
data Maybe a = Nothing | Just a

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

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 let delta = b * b - 4 * a * c
                  rDelta <- radical delta
                  return (negate b + rDelta) / (2 * a)
```

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.

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.

```
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 let delta = b * b - 4 * a * c
                  rDelta <- radical delta
                  return (negate b + rDelta) / (2 * a)
```

Monada Reader (stare nemodificabilă)

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

ask :: Reader env env
ask = Reader id
```

Monada Reader (stare nemodificabilă)

```haskell
newtype Reader env a = Reader { runReader :: 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 a = runReader ma env
                  in  runReader (k a) env
```

Monada Reader — exemplu: mediu de evaluare

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

data Prop ::= Var String | Prop :&: Prop
type Env = [(String, Bool)]

var :: String -> Reader Env Bool
var x = do
  env <- ask
  fromMaybe False (lookup x env)

eval :: Prop -> Reader Env Bool
eval (Var x) = var x
eval (p1 :&: p2) = do
  b1 <- eval p1
  b2 <- eval p2
  return (b1 && b2)
```

Monada `Writer` (varianta lungă)

Clasele Semigrup și Monoid

Clasa de tipuri Semigroup

O mulțime, cu o operație <> care ar trebui să fie asociativă

```
class Semigroup a where
  (<>) :: a -> a -> a
```

Clasa de tipuri Monoid

Un semigrup cu unitatea `mempty`. `mappend` este alias pentru <>.

```
class Semigroup a => Monoid a where
  mempty :: a
  mappend :: a -> a -> a
  mappend = (<>)
```

Foarte multe tipuri sunt instanțe ale lui Monoid. Exemplul clasic: listele.

Monada Writer (varianta lungă)

```haskell
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)
```

Functor vs Applicative vs Monad

Functor vs Applicative : Efecte laterale

Functor

Schimbă rezultatul: efectele laterale rămân aceleași

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

Applicative

Combină computații independente, propagând efectele laterale

```
class Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

O utilizare des întâlnită a operatorilor aplicativi e pentru a combina computații cu efecte printr-o funcție pură

```
eval :: Prop -> Reader Env Bool
eval (Var x) = var x
eval (p1 :&: p2) = pure (&&) <*> eval p1 <*> eval p2
```

Monad vs Applicative

Applicative — combinarea independentă a computațiilor

```
eval :: Prop -> Reader Env Bool
eval (Var x) = var x
eval (p1 :&: p2) = pure (&&) <*> eval p1 <*> eval p2

rnd2 :: State Word32 Word32
rnd2 = pure (+) <*> rnd <*> rnd
```

Monad — secvențierea computațiilor

Definește continuări, care depind de rezultatul computațiilor precedente.

```
sau :: a -> a -> State Word32 a
x `sau` y = do
  r <- rnd
  return (if r >= maxBound `div` 2 then y else x)
```

Functor și Applicative pot fi 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 F where                    -- ma >>= \a -> return (f
  fmap f ma = pure f <*> ma                 -- ma >>= (return . f)
```

Pe săptămâna viitoare!