

Programare declarativă

Introducere în programarea funcțională folosind Haskell

Ioana Leuștean
Ana Cristina Turlea

Departamentul de Informatică, FMI, UB
ioana@fmi.unibuc.ro
ana.turlea@fmi.unibuc.ro

Monada State

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
```

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

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

În Haskell, monada este o clasă de tipuri!

Monada State

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

```
instance Monad (State state) where
```

```
    return va = State (\s -> (va, 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
```

```
-- h :: state -> (b, state)
```

```
-- ma >=> k :: State state b
```

Monada State

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



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

Funcții ajutătoare:

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



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

Semantica unui mini calculator

Limbajul unui mini calculator

Definim în Haskell limbajul unui mini calculator:

```
data Prog  = On Instr
data Instr = Off | Expr :> Instr
data Expr  = Mem | V Int | Expr :+ Expr
```

Semantica în limbaj natural

Dorim ca un program să afișeze lista valorilor corespunzătoare expresiilor, unde Mem reprezintă ultima valoare calculată. Valoarea inițială a lui Mem este 0.

De exemplu, programul

```
On ((V 3) :> ((Mem :+ (V 5)) :> Off))
```

va afișa lista [3,8]

Semantica denotațională - domenii semantice

Domeniile semantice în Haskell

```
type Env = Int    -- valoarea celulei de memorie  
type DomProg = [Int]  
type DomInstr = Env -> [Int]  
type DomExpr = Env -> Int
```

Semantica denotațională

Pentru a defini semantica denotațională trebuie să evaluăm (interpretăm) categoriile sintactice în domeniile semantice corespunzătoare.

Interpretări (Evaluări)

```
prog  :: Prog -> DomProg  
stmt  :: Instr -> DomInstr  
expr  :: Expr -> DomExpr
```


Semantica denotațională folosind monada State

```
data Program  = On Statements
data Statements = Off | Expression :> Statements
data Expression = Mem | V Int | Expression :+ Expression

type Env = Int
type Val = Int

prog :: Program -> State Env [Val]
stmt :: Statements -> State Env [Val]
expr :: Expression -> State Env Val
```

Semantica denotațională folosind monada State

```
data Program  = On Statements
```

```
type Env = Int
```

```
type Val = Int
```

```
prog :: Program -> State Env [Val]
```

```
prog (On s) = do
```

```
    put 0
```

```
    stmt s
```

Semantica denotațională folosind monada State

```
data Statements = Off | Expression :> Statements
```

```
type Env = Int
```

```
type Val = Int
```

```
stmt :: Statements -> State Env [Val]
```

```
stmt (e :> s) = do
```

```
    v <- expr e
```

```
    vs <- (put v >> stmt s)
```

```
    return (v:vs)
```

```
stmt Off = return []
```

Semantica denotațională folosind monada State

```
data Expression = Mem | V Int | Expression :+: Expression
```

```
type Env = Int
```

```
type Val = Int
```

```
expr :: Expression -> State Env Val
```

```
expr (e1 :+: e2) = do
    v1 <- expr e1
    v2 <- expr e2
    return (v1 + v2)
```

```
expr (V n) = return n
```

```
expr Mem = get
```

Semantica denotațională folosind monada State

```
test :: Program -> ([Val], Env)
test vp = runState (prog vp) $ 0
```

```
vprog = On ((V 3) :> (( Mem :+ (V 5)) :> Off))
```

Jocul Nim

Jocul Nim

În continuare vom implementa jocul Nim:

- joacă mai mulți jucători care mută pe rând,
- **Tabla:** mai multe gramezi cu piese,
- **Mutare:** jucatorul ia una sau mai multe piese dintr-o gramadă (poate lua toată grămada, dar piesele trebuie luate din aceeași grămadă),
- **Regula:** câștigă jucătorul care a făcut ultima mutare.

Jocul Nim - Exemplu

Tabla: [2,5,4]

Un șir de mutări posibile:

Alice ia 4 obiecte din gramada 2

Bob ia 1 obiect din gramada 3

Alice ia 2 obiecte din gramada 1

Bob ia 1 obiect din gramada 2

Alice ia 3 obiecte din gramada 3

Jocul Nim - reprezentarea datelor

```
import Data.Maybe (fromJust)

type HeapName = Int    -- numarul gramezii (incepe de la 1)
type HeapVal  = Int    -- numarul de elemente
type Board    = [HeapVal] -- lista gramezilor
type Player   = String -- jucatorul care a facut mutarea

type Game = [(Player, Turn)]
type Turn = (HeapName, HeapVal) -- (x,y)
                                   -- din gramada x iau y obiecte
```

Jocul Nim - exerciții pregătitoare

Scrieți o funcție care face o mutare pe o tablă

```
oneTurn :: Turn -> Board -> Maybe Board
oneTurn (n, v) b
  | n > length b = Nothing
  | otherwise =
      let (l1, v':l2) = splitAt (n - 1) b
      in  if v > v'
          then Nothing
          else Just (l1 ++ (v' - v):l2)
```

```
> oneTurn (2,3) [1,4,5]
Just [1,1,5]
```

Jocul Nim - exerciții pregătitoare

Reprezentarea jucătorilor și a ordinii în care mută

```
next :: Player -> Player  
next "Bob" = "Alice"  
next "Alice" = "Bob"
```

Presupunem că avem numai doi jucători

Jocul Nim - exerciții pregătitoare

Generați toate mutările posibile pentru o tablă dată

```
allTurns :: Board -> [Turn]
allTurns xs = [(i,x) | (i,z) <- (zip [1..] xs), x<-[1..z]]

> allTurns [2,3]
[(1,1),(1,2),(2,1),(2,2),(2,3)]

nextTurns :: Board -> [Board]
nextTurns b = [b' | Just b' <- map ('oneTurn' b)(allTurns b)]

> nextTurns [2,3]
[[1,3],[0,3],[2,2],[2,1],[2,0]]
```

Jocul Nim - arborele jocului

Construim arborele jocului pentru o tablă dată astfel:

- nodurile arborelui sunt table
- fiii unui nod sunt tablele la care se ajunge făcând o singură mutare
- frunzele sunt table care nu au fii (în frunze vor fi liste care conțin numai '0')

```
data GameTree = V Board [GameTree]  
  deriving Show
```

Jocul Nim - arborele jocului

```

gameTree :: Board -> GameTree
gameTree b = V b subTrees
  where
    subTrees = [gameTree b' | b' <- nextTurns b]

> gameTree [2,1]
V [2,1] [V [1,1] [V [0,1] [V [0,0] []] ,
            V [1,0] [V [0,0] []]] ,
        V [0,1] [V [0,0] []] ,
        V [2,0] [V [1,0] [V [0,0] []] , V [0,0] []]]

```

Jocul Nim - variante câștigătoare

Pentru fiecare jucător calculați numărul de variante câștigătoare

- cu 1 numărăm 'null' pe nivelele pare 0, 2, ... din arborele jocului
- cu 0 numărăm 'null' pe nivelele impare 1,3, ... din arborele jocului

```
wins :: Int -> GameTree -> Int
wins 1 (V _ []) = 1  -- frunza se numara
wins 0 (V _ []) = 0  -- frunza nu se numara
wins 0 (V _ xs) = sum (map (wins 1) xs)
wins 1 (V _ xs) = sum (map (wins 0) xs)

-- de cate ori castiga primul jucator
firstWins, secondWins :: Board -> Int
firstWins board = wins 0 (gameTree board)
secondWins board = wins 1 (gameTree board)
```

```
> firstWins [2,5,6]
518594
```

Jocul Nim - structura generală

- Starea jocului este formată numai din lista grămezilor:

```
type GameState = Board
```

- Vom folosi monada 'State GameState' pentru a menține starea jocului:

```
playGame :: Board -> State GameState Game
```

- Structura generală a programului:

```
showGame :: State GameState Game -> String
```

```
playGame :: Board -> State GameState Game
```

```
gameBoard = [ 2, 5, 4]
```

```
game = do
```

```
    putStrLn ("Board: " ++ (show gameBoard))
```

```
    putStrLn $ showGame . playGame $ gameBoard
```


Jocul Nim - 'showGame'

Afișează șirul de mutări

```
showGame :: State GameState Game -> String
```

```
showGame g = showG a
```

```
  where (a, _) = runState g []
```

```
showG :: Game -> String
```

```
showG [] = ""
```

```
showG ((w, (x ,v)) : ws) =
```

```
  w ++ " ia " ++ (show v)
```

```
  ++ " obiecte din gramada " ++ (show x)  ++ "\n"
```

```
  ++ (showG ws)
```

Jocul Nim - 'playGame'

În această variantă fiecare jucător are o strategie

```

playGame :: Board -> State GameState Game
playGame initb = do
    put initb -- seteaza starea initiala
    loop "Alice" -- Alice muta prima

loop w = do
    turn <- playWithStrat (strategy w)
    -- executa mutarea data de strategie
    -- si o intoarce pentru oprire/afisare
    if turn == Nothing then return []
    else do
        turn1 <- loop (next w)
        -- trece la urmatorul jucator
        return $ [(w, fromJust turn)] ++ turn1
  
```

Jocul Nim - 'playWithStrat'

```
strategy :: Player -> (Board -> Turn)
```

```
playWithStrat :: (Board -> Turn) -> State GameState (Maybe Turn)
```

```
playWithStrat strat =
```

```
  do
```

```
    currentBoard <- get
```

```
    if all (==0) currentBoard
```

```
      then return Nothing -- jocul s-a incheiat
```

```
      else do
```

```
        let
```

```
          turn = strat currentBoard
```

```
          Just newBoard = oneTurn turn currentBoard
```

```
          -- mutarea este data de strategie
```

```
        put newBoard
```

```
        return (Just turn)
```

Starea jocului este menținută de monada State GameState

let în interiorul do

- în interiorul unui "bloc" **do** poate fi folosită expresia **let** pentru a instanția variabilele folosite în expresiile monadice următoare.

```
sayHi = do
    putStr "What is your first name? "
    first <- getLine
    putStr "And your last name? "
    last <- getLine
    let full = first++" "++last
    putStrLn ("Hi, "++full++"!")
```

```
> sayHi
What is your first name? Ioana
And your last name? Leustean
Hi, Ioana Leustean!
```

Jocul Nim - strategii

Mutarea este determinata de o strategie

```
strategy :: Player -> (Board -> Turn)
strategy "Alice" = strat1
strategy "Bob"   = strat2
```

Strategiile sunt foarte simple

```
strat1 xs = (head pozitii, 1)
  where pozitii = [p | (p, y) <- zip [1..] xs, y > 0]
```

```
strat2 xs = (last pozitii, 1)
  where pozitii = [p | (p, y) <- zip [1..] xs, y > 0]
```

```
strat3 xs = (head pozitii, xs !! (head pozitii - 1))
  where pozitii = [p | (p, y) <- zip [1..] xs, y > 0]
```

Exemplu de rulare cu 'strat1' si 'strat2'

```
> game
```

```
Board: [2,5,4]
```

```
Alice ia 1 obiecte din gramada 1
```

```
Bob ia 1 obiecte din gramada 3
```

```
Alice ia 1 obiecte din gramada 1
```

```
Bob ia 1 obiecte din gramada 3
```

```
Alice ia 1 obiecte din gramada 2
```

```
Bob ia 1 obiecte din gramada 3
```

```
Alice ia 1 obiecte din gramada 2
```

```
Bob ia 1 obiecte din gramada 3
```

```
Alice ia 1 obiecte din gramada 2
```

```
Bob ia 1 obiecte din gramada 2
```

```
Alice ia 1 obiecte din gramada 2
```

Exemplu de rulare cu 'strat3' si 'strat2'

```
*Main> game
```

```
Board: [2,5,4]
```

```
Alice ia 2 obiecte din gramada 1
```

```
Bob ia 1 obiecte din gramada 3
```

```
Alice ia 5 obiecte din gramada 2
```

```
Bob ia 1 obiecte din gramada 3
```

```
Alice ia 2 obiecte din gramada 3
```

Pe săptămâna viitoare!