Curs 11

2021-2022

Fundamentele limbajelor de programare

Cuprins

Implementarea Mini-Haskell în Haskell

Mini-Haskell

Vom defini folosind Haskell un mini limbaj funcţional şi un interpretor pentru acesta.

Mini-Haskell

Vom defini folosind Haskell un mini limbaj funcţional şi un interpretor pentru acesta.

- Limbajul Mini-Haskell conţine:
  - expresii de tip Int
  - expresii de tip funcţie (λ-expresii)
  - expresii provenite din aplicarea functiilor

Mini-Haskell

Vom defini folosind Haskell un mini limbaj funcţional şi un interpretor pentru acesta.

- Limbajul Mini-Haskell conţine:
  - expresii de tip Int
  - expresii de tip funcţie (λ-expresii)
  - expresii provenite din aplicarea functiilor
- Pentru a defini semantica limbajului vom introduce domeniile semantice (valorile) asociate expresiilor limbajului.
- Pentru a evalua (interpreta) expresiile vom defini un mediu de evaluare in care vom retine variabilele si valorile curente asociate.

Mini-Haskell (λ-calcul cu întregi). Sintaxă

```
type Name = String

data Term = Var Name
          | Con Integer
          | Term :+: Term
          | Lam Name Term
          | App Term Term
deriving (Show)
```

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)
```

Ce λ-expresie defineste termenul de mai sus?

Domenii

Domeniul valorilor

```
data Value = Num Integer
           | Fun (Value -> Value)
           | Wrong -- pentru reprezentarea erorilor
```

Mediul de evaluare

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

Domeniul de evaluare

Fiecărei expresii i se va asocia ca denotație o funcție de la medii de evaluare la valori:

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

Afișarea expresiilor

```
instance Show Value where
  show (Num x) = show x
  show (Fun _) = "<function>"
  show Wrong   = "<wrong>"
```

Observație

Funcțiile nu pot fi afișate ca atare, ci doar generic.

Evaluarea expresiilor Mini-Haskell în Haskell

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

interp (Con i) _ =  Num i

interp (t1 :+: t2) env = add (interp t1 env) (interp t2 env)

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

Evaluarea expresiilor Mini-Haskell în Haskell

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

interp (Var x) env = lookupM x env

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

-- lookup din modulul Data.List
lookup :: (Eq a) => a -> [(a,b)] -> Maybe b
lookup  key []          =  Nothing
lookup  key ((x,y) : xys)
          | key == x    =  Just y
          | otherwise   =  lookup key xys
```

Evaluarea expresiilor Mini-Haskell în Haskell

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

interp (Lam x e) env = Fun $ \ v -> interp e ((x,v):env)
```

Evaluarea expresiilor Mini-Haskell în Haskell

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

interp (Lam x e) env = Fun $ \ v -> interp e ((x,v):env)

interp (App t1 t2) env = apply f v
  where
      f = interp t1 env
      v = interp t2 env

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

Implementarea Mini-Haskell în Haskell

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

interp (Var x) env = lookupM x env
     where lookupM x env = case lookup x env of
                             Just v  -> v
                             Nothing -> Wrong

interp (Con i) _ =  Num i

interp (t1 :+: t2) env = add (interp t1 env) (interp t2 env)
      where add (Num i) (Num j) = Num $ i + j
            add  _ _            =  Wrong

interp (Lam x e) env =  Fun $ \ v -> interp e ((x,v):env)

interp (App t1 t2) env = apply (interp t1 env) (interp t2 env)
      where apply (Fun k) v = k v
            apply _ _       =  Wrong
```

Evaluarea expresiilor Mini-Haskell în Haskell

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

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

*Main> test pgm
"42"
```

Evaluarea expresiilor Mini-Haskell în Haskell

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

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

*Main> test pgmW
"<wrong>"
```

Evaluarea expresiilor Mini-Haskell în Haskell

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

testIO :: Term -> IO ()
testIO t =  putStrLn . show $ interp t []

> test pgm
42
```

Evaluarea expresiilor Mini-Haskell în Haskell

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

testIO :: Term -> IO ()
testIO t =  putStrLn . show $ interp t []

> test pgm
42
```

Ce este IO?

Monade în Haskell

Haskell

- În loc să modificăm datele existente, calculăm valori noi din valorile existente, folosind funcții.
- Funcțiile sunt pure: aceleași rezultate pentru aceleași intrări.
- Puritatea asigură consistență:
  - O bucată de cod nu poate corupe datele altei bucăți de cod.
  - Mai ușor de testat decât codul care interacționează cu mediul.

Haskell

- În loc să modificăm datele existente, calculăm valori noi din valorile existente, folosind funcții.
- Funcțiile sunt pure: aceleași rezultate pentru aceleași intrări.
- Puritatea asigură consistență:
  - O bucată de cod nu poate corupe datele altei bucăți de cod.
  - Mai ușor de testat decât codul care interacționează cu mediul.

Cum interacționăm cu mediul extern, păstrând puritatea?

Haskell

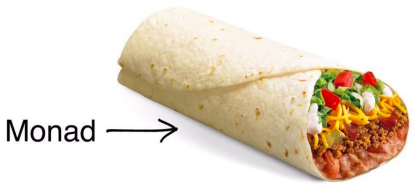
- În loc să modificăm datele existente, calculăm valori noi din valorile existente, folosind funcții.
- Funcțiile sunt pure: aceleași rezultate pentru aceleași intrări.
- Puritatea asigură consistență:
  - O bucată de cod nu poate corupe datele altei bucăți de cod.
  - Mai ușor de testat decât codul care interacționează cu mediul.

Cum interacționăm cu mediul extern, păstrând puritatea?
Folosim *monade*!

Ce este o monadă?

Există multe răspunsuri, variind între

- O monadă este un burrito. `https://byorgey.wordpress.com/2009/01/12/abstraction-intuition-and-the-monad-tutorial-fallacy/`

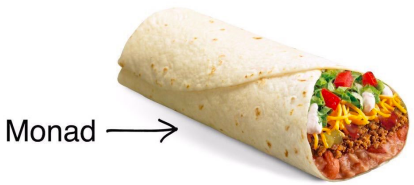


`https://twitter.com/monadburritos`

Ce este o monadă?

Există multe răspunsuri, variind între

- O monadă este un burrito. `https://byorgey.wordpress.com/2009/01/12/abstraction-intuition-and-the-monad-tutorial-fallacy/`



`https://twitter.com/monadburritos`

- "All told, a monad in X is just a monoid in the category of endofunctors in X, with product x replaced by composition of endofunctors and unit set by the identity endofunctor."

Saunders Mac Lane, Categories for the Working Mathematician, 1998.

Funcţii îmbogăţite şi efecte

- Funcţie simplă: $x \mapsto y$

  ştiind x, obţinem direct y

Funcţii îmbogăţite şi efecte

- Funcţie simplă: $x \mapsto y$

  ştiind x, obţinem direct y

- Funcţie îmbogăţită: $x \mapsto$

ştiind x, putem să extragem y şi producem un efect

https://bartoszmilewski.com/2016/11/21/monads-programmers-definition/

https://bartoszmilewski.com/2016/11/30/monads-and-effects/

Funcţii îmbogăţite şi efecte

Funcţie îmbogăţită: $x \mapsto$ 

Exemplu

- Folosind tipul Maybe a

```
data Maybe a = Nothing | Just a
f :: Int -> Maybe Int
f x = if x < 0 then Nothing else (Just x)
```

Funcţii îmbogăţite şi efecte

Funcţie îmbogăţită: $x \mapsto$ 

Exemplu

- Folosind un tip care are ca efect un mesaj

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

f :: Int -> Writer String Int
f x = if x < 0 then (Writer (-x, "negativ"))
               else (Writer (x, "pozitiv"))
```

Logging în Haskell

„Îmbogățim” rezultatul funcției cu mesajul de log.

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

Observații

- 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")

Compunerea funcţiilor

- Principala operaţie pe care o facem cu funcţii este compunerea

```
f :: a -> b , g :: b -> c , g . f :: a -> c
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Valoarea de tip b este transmisă direct funcţiei g.

Compunerea funcţiilor

- Principala operaţie pe care o facem cu funcţii este compunerea

```
f :: a -> b , g :: b -> c , g . f :: a -> c
(.) :: (b -> c) -> (a -> b) -> a -> c
```

  Valoarea de tip b este transmisă direct funcţiei g.

- Ce facem dacă

```
f :: a -> m b , g :: b -> m c
```

  unde m este un constructor de tip care îmbogăţeşte tipul?

Compunerea funcţiilor

- Principala operaţie pe care o facem cu funcţii este compunerea

```
f :: a -> b , g :: b -> c , g . f :: a -> c
(.) :: (b -> c) -> (a -> b) -> a -> c
```

  Valoarea de tip b este transmisă direct funcţiei g.

- Ce facem dacă

```
f :: a -> m b , g :: b -> m c
```

  unde m este un constructor de tip care îmbogăţeşte tipul?

  De exemplu,
  - m = Maybe
  - m = Writer log

- Atenţie! m trebuie să aibă un singur argument.

Compunerea funcţiilor

```
f :: a -> m b , g :: b -> m c
```

unde m este un constructor de tip care îmbogăţeşte tipul.

Vrem să definim o "compunere" pentru funcţii îmbogăţite

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

Atunci când definim g <=< f trebuie să extragem valoarea întoarsă de f şi să o trimitem lui g.

Exemplu: logging în Haskell

„Îmbogățim” rezultatul funcției cu mesajul de log.

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

logInc :: Int -> Writer String Int
logInc x =
   Writer (x + 1, "Called inc with arg " ++ show x ++ "\n")
```

Problemă: Cum calculăm logInc (logInc x)?

Exemplu: logging în Haskell

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

logInc :: Int -> Writer String Int
logInc x =
   Writer (x + 1, "Called inc with arg " ++ show x ++ "\n")


logInc2 :: Int -> Writer String Int
logInc2 x = let (y, log1) = runWriter (logInc x)
                (z, log2) = runWriter (logInc y)
            in  Writer (z, log1 ++ log2)
```

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.

Exemplu

```
> logInc x = Writer
(x + 1, "Called inc with arg " ++ show x ++ "\n")

> logInc <=< logInc $ 3
Writer {runWriter =
(5,"Called inc with arg 3\n Called inc with arg 4\n")}
```

Observaţie: Funcţia (<=<) este definită în Control.Monad

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 -> m b este tipul continuărilor / a funcțiilor cu efecte laterale
- >>= este operația de „secvențiere” a computațiilor

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 -> m b este tipul continuărilor / a funcțiilor cu efecte laterale
- >>= este operația de „secvențiere” a computațiilor
- în Control.Monad sunt definite
  - f >=> g = \x -> f x >>= g
  - (<=<) = flip (>=>)

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 -> m b este tipul continuărilor / a funcțiilor cu efecte laterale
- >>= este operația de „secvențiere” a computațiilor
- în Control.Monad sunt definite
  - f >=> g = \x -> f x >>= g
  - (<=<) = flip (>=>)

Applicative va fi discutată mai târziu

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 -> m b este tipul continuărilor / a funcțiilor cu efecte laterale
- >>= este operația de „secvențiere” a computațiilor

În Haskell, monada este o clasă de tipuri!

Exemple: monada Maybe

```
lookup :: Eq a => a -> [(a, b)] -> Maybe b

> (lookup 3 [(1,2), (3,4)]) >>=
  (\x -> if (x<0) then Nothing else (Just x))
Just 4

> (lookup 3 [(1,2), (3,-4)]) >>=
  (\x -> if (x<0) then Nothing else (Just x))
Nothing

> (lookup 3 [(1,2)]) >>=
  (\x -> if (x<0) then Nothing else (Just x))
Nothing
```

Proprietățile monadelor

Asociativitate şi element neutru

Operația <=< de compunere a funcţiilor îmbogăţite este asociativă şi are element neutru return.

Proprietățile monadelor

Asociativitate şi element neutru

Operația <=< de compunere a funcţiilor îmbogăţite este asociativă şi are element neutru return.

- Element neutru (la dreapta): g <=< return = g

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

- Element neutru (la stânga): return <=< g = g

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

- Asociativitate: h <=< (g <=< f) = (h <=< g) <=< f

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

Notația do pentru monade

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

Notația do pentru monade

Notaţia cu operatori	Notaţia 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

Notaţia cu operatori	Notaţia 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

De exemplu

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

devine

Notația do pentru monade

De exemplu

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

devine

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

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 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 va  >>= k   = k va
  Nothing  >>= _   = Nothing
```

Monada Maybe (a funcțiilor parțiale)

```
data Maybe a = Nothing | Just a

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

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

Monada Maybe (a funcțiilor parțiale)

```
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)

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

Monada Either (a excepțiilor)

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

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

Monada Either (a excepțiilor)

```
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
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

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 = (<>)
```

Monada Writer

Clasa de tipuri Functor

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

Clasa de tipuri Applicative

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

Monada Writer

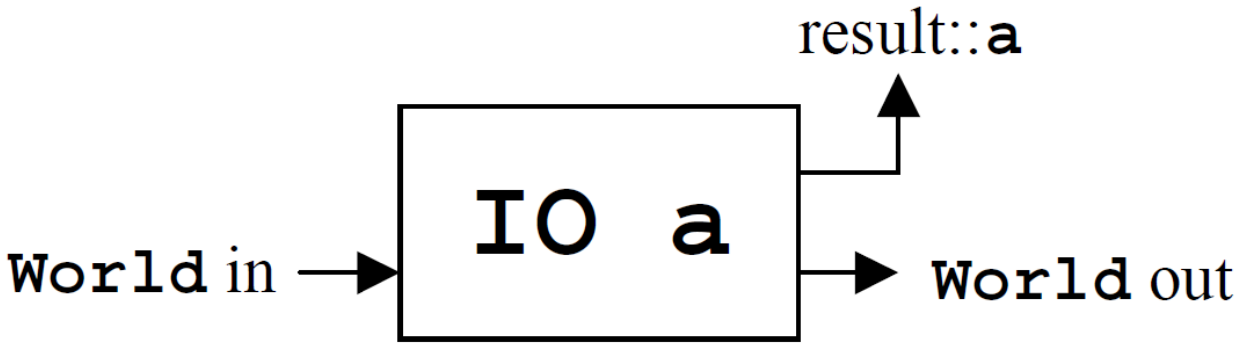
```
newtype Writer log a = Writer {runWriter :: (a, log)}

instance Functor (Writer log) where
    fmap f (Writer (a,log)) = Writer (f a,log)

instance Monoid log => Applicative (Writer log) where
    pure a = Writer (a,mempty)
    (Writer (f,log1)) <*> (Writer (a,log2)) =
            Writer (f a, log1 <> log2)

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

Monada Writer - Exemplu logging

```
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 IO

```
type IO a = RealWorld -> (a, RealWorld)
```



S. Peyton-Jones, Tackling the Awkward Squad: ...

Comenzi cu valori

- IO () corespunde comenzilor care nu produc rezultate

```
putChar :: Char -> IO ()
putStr  :: String -> IO ()
putStrLn :: String -> IO ()
```

- În general, IO a corespunde comenzilor care produc rezultate de tip a.

```
getChar :: IO Char
getLine  :: IO String
```

Pe săptămâna viitoare!