Functional Programming
Monadic Prelude

Jevgeni Kabanov

Department of Computer Science
University of Tartu
Introduction

Previously on Functional Programming

- Monadic laws
- Monad class (\(\gg=\) and \(\text{return}\))
- MonadPlus class (\(\text{mzero}\) and \(\text{mplus}\))
- do-notation
- Maybe monad
- List monad
- State monad
- IO monad
Overview

1. List functions
2. Conditionals
3. Lifting
4. MonadPlus functions
Introduction

Overview

- Monad power comes from very high degree of abstraction

- Haskell comes with a library of functions that are defined across all monads

- These functions correspond to control structures in most imperative languages

- In fact given the do-notation and these functions we can program in Haskell using imperative approach
Introduction

Overview

- Monad power comes from very high degree of abstraction

- Haskell comes with a library of functions that are defined across all monads

- These functions correspond to control structures in most imperative languages

- In fact given the do-notation and these functions we can program in Haskell using imperative approach
Introduction

Overview

- Monad power comes from very high degree of abstraction

- Haskell comes with a library of functions that are defined across all monads

- These functions correspond to control structures in most imperative languages

- In fact given the do-notation and these functions we can program in Haskell using imperative approach
Introduction

Overview

- Monad power comes from very high degree of abstraction.

- Haskell comes with a library of functions that are defined across all monads.

- These functions correspond to control structures in most imperative languages.

- In fact given the do-notation and these functions we can program in Haskell using imperative approach.
Outline

1. List functions
2. Conditionals
3. Lifting
4. MonadPlus functions
sequence definition

\[
\text{sequence\_} :: \text{Monad}\ m \Rightarrow [m\ a] \rightarrow m\ ()
\]
\[
\text{sequence\_} = \text{foldr}\ (\gg\gg)\ (\text{return}\ ())
\]

\[
\text{sequence} :: \text{Monad}\ m \Rightarrow [m\ a] \rightarrow m\ [a]
\]
\[
\text{sequence} = \text{foldr}\ m\text{cons}\ (\text{return}\ [])
\]

where \( m\text{cons}\ p\ q = \)
\[
p \gg\gg \lambda x \rightarrow q \gg\gg \lambda y \rightarrow \text{return}\ (x : y)
\]
sequence example

Monads> sequence [print 1, print 2, print 'a']
1
2
'a'

*Monads> it
[(),(),()]

*Monads> sequence_ [print 1, print 2, print 'a']
1
2
'a'

*Monads> it
()
sequence example 3

Prelude> sequence [Just 1, Just 2, Nothing, Just 3]
sequence

sequence example 3

Prelude> sequence [Just 1, Just 2, Nothing, Just 3]
Nothing

Maybe is asymmetrical with respect to nothing!
mapM

mapM definition

\[
\text{mapM} :: \text{Monad } m \Rightarrow (a \rightarrow m b) \rightarrow [a] \rightarrow m [b]
\]
\[
\text{mapM} f \ as = \text{sequence} (\text{map} \ f \ as)
\]

\[
\text{mapM} _\_ :: \text{Monad } m \Rightarrow (a \rightarrow m b) \rightarrow [a] \rightarrow m ()
\]
\[
\text{mapM} _\_ \ f \ as = \text{sequence} _\_ (\text{map} \ f \ as)
\]
mapM

**mapM example 1**

Monads> mapM_ print [1,2,3,4,5]
1
2
3
4
5

**mapM example 2**

```
putString :: [Char] -> IO ()
putString s = mapM_ putChar s
```
forM

forM definition

\[
\text{forM} :: \text{Monad } m \Rightarrow [a] \rightarrow (a \rightarrow m\ b) \rightarrow m\ [b] \\
\text{forM} = \text{flip mapM} \\
\text{forM}_\_ :: \text{Monad } m \Rightarrow [a] \rightarrow (a \rightarrow m\ b) \rightarrow m\ () \\
\text{forM}_\_ = \text{flip mapM}_\_
\]

forM_ example

\[
\text{main} = \text{do} \\
\quad \text{forM}_\_ [1..10] (\lambda i \rightarrow \text{print } i)
\]
**filterM**

**filterM definition**

\[ \text{filterM} :: \text{Monad } m \Rightarrow (a \to m \text{ Bool}) \to [a] \to m [a] \]

\[
\text{filterM } p \; [] = \text{return } [] \\
\text{filterM } p \; (x : xs) = \text{do } b \leftarrow p \; x \\
\quad \text{ys } \leftarrow \text{filterM } p \; xs \\
\quad \text{return } (\text{if } b \text{ then } (x : \text{ys}) \text{ else } \text{ys})
\]

**filterM example**

\[
\text{main } = \text{do } \\
\quad \text{names } \leftarrow \text{getArgs} \\
\quad \text{dirs } \leftarrow \text{filterM } \text{doesDirectoryExist } \text{names} \\
\quad \text{mapM } \_ \text{ putStrLn } \text{dirs}
\]
**foldM**

### foldM definition

\[
\text{foldl} :: (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow a \\
\text{foldl} f z [] = z \\
\text{foldl} f z (x : xs) = \text{foldl} f (f z x) xs
\]

\[
\text{foldM} :: (\text{Monad } m) \Rightarrow (a \rightarrow b \rightarrow m a) \rightarrow a \rightarrow [b] \rightarrow m a \\
\text{foldM} f a [] = \text{return } a \\
\text{foldM} f a (x : xs) = f a x >>= \lambda y \rightarrow \text{foldM} f y xs
\]

Note that we lift the result of all functions under the monad.
foldM

foldM explanation

\[
\text{foldM } f \ a1 \ [x1, x2, ..., xn] = \textbf{do}
\]
\[
a2 \leftarrow f \ a1 \ x1
\]
\[
a3 \leftarrow f \ a2 \ x2
\]
\[
...
\]
\[
f \ an \ xn
\]
foldM example

Monads> foldM (\a b ->
    putStrLn (show a ++ "+" ++ show b ++
      "=" ++ show (a+b)) »
    return (a+b)) 0 [1..5]

0+1=1
1+2=3
3+3=6
6+4=10
10+5=15

Monads> it
15
foldM

foldM example 2

data Sheep = Sheep { name :: String,  
mother :: Maybe Sheep, father :: Maybe Sheep }  
dolly :: Sheep  
dolly = let  
    adam = Sheep "Adam" Nothing Nothing  
    eve = Sheep "Eve" Nothing Nothing  
    uranus = Sheep "Uranus" Nothing Nothing  
    gaea = Sheep "Gaea" Nothing Nothing  
    kronos = Sheep "Kronos" (Just gaea) (Just uranus)  
    holly = Sheep "Holly" (Just eve) (Just adam)  
    roger = Sheep "Roger" (Just eve) (Just kronos)  
    molly = Sheep "Molly" (Just holly) (Just roger)  
in Sheep "Dolly" (Just molly) Nothing
foldM

foldM example 2

traceFamily ::
    [(Sheep → Maybe Sheep)] → Sheep → Maybe Sheep
traceFamily l s = foldM (flip ($)) s l

paternalGrandmother =
    traceFamily [father, mother]

mothersPaternalGrandfather =
    traceFamily [mother, father, father]

Output:

*Main> paternalGrandmother dolly
Nothing
*Main> mothersPaternalGrandfather dolly
Just "Kronos"
foldM

Map definition

data Map k a

empty :: Map k a
insert :: Ord k ⇒ k → a → Map k a → Map k a
lookup :: (Monad m, Ord k) ⇒ k → Map k a → m a
toList :: Map k a → [(k, a)]
foldM

foldM example 3

```
data Entry = Entry{key :: String, value :: String}
type Dict = Map String String

addEntry :: Dict → Entry → Dict
addEntry d e = insert (key e) (value e) d

addDataFromFile :: Dict → Handle → IO Dict
addDataFromFile dict hd l = do
    contents ← hGetContents hd l
    entries ← return (map read (lines contents))
    return (foldl (addEntry) dict entries)
```
foldM

foldM example 3

\[
\text{main} :: \text{IO (}) \\
\text{main} = \text{do} \\
\text{files} \leftarrow \text{getArgs} \\
\text{handles} \leftarrow \text{mapM openForReading files} \\
\text{dict} \leftarrow \text{foldM addDataFromFile empty handles} \\
\text{print (toList dict)}
\]
join

**join definition**

$join :: (Monad m) \Rightarrow m (m a) \rightarrow m a$

$join \ x = x >= id$

Note that $x >= f = (join \circ fmap f) \ x$.

**join example**

Monads> join (Just (Just 'a'))
Just 'a'

Monads> join (return (putStrLn "hello"))
hello

Monads> return (putStrLn "hello")

Monads> join [[1,2,3],[4,5]]
[1,2,3,4,5]
Outline

1. List functions
2. Conditionals
3. Lifting
4. MonadPlus functions
**when**

**when and unless definition**

\[
\text{when} :: (\text{Monad } m) \Rightarrow \text{Bool} \rightarrow m () \rightarrow m ()
\]

\[
\text{when } p \ s = \text{if } p \text{ then } s \text{ else return } ()
\]

\[
\text{unless} :: (\text{Monad } m) \Rightarrow \text{Bool} \rightarrow m () \rightarrow m ()
\]

\[
\text{unless } p \ s = \text{when } (\neg p) \ s
\]

**when example**

\[
\text{Monads> mapM } _\ (\backslash l \rightarrow \text{when } (\text{not } \$ \text{ null } l) \ (\text{putStrLn } l))
\
["","abc","def","","","","ghi"]
\]

abc
def
ghi
Outline

1. List functions
2. Conditionals
3. Lifting
4. MonadPlus functions
**liftM**

**liftM and liftM2 definition**

\[
\text{liftM} :: (\text{Monad } m) \Rightarrow (a \to b) \to (m a \to m b)
\]

\[
\text{liftM} \; f = \lambda a \to \text{do} \{ a' \leftarrow a; \text{return} \; (f \; a') \}
\]

\[
\text{liftM2} :: (\text{Monad } m) \Rightarrow (a \to b \to c) \to (m a \to m b \to m c)
\]

\[
\text{liftM2} \; f = \lambda a \; b \to \text{do} \{ a' \leftarrow a; b' \leftarrow b; \text{return} \; (f \; a' \; b') \}
\]

- Lifting allows to apply pure functions point-free to monadic values
- Together with monadic bind it constitutes a functional approach as apposed to the do-notation
\textit{liftM}

\textit{liftM} and \textit{liftM2} definition

\begin{align*}
\text{liftM} :: (\text{Monad } m) \Rightarrow (a \to b) \to (m\ a \to m\ b) \\
\text{liftM } f &= \lambda a \rightarrow \text{do } \{ a' \leftarrow a; \text{return } (f\ a') \} \\
\text{liftM2} :: (\text{Monad } m) \Rightarrow \\
&\quad (a \to b \to c) \to (m\ a \to m\ b \to m\ c) \\
\text{liftM2 } f &= \\
&\quad \lambda a\ b \rightarrow \text{do } \{ a' \leftarrow a; b' \leftarrow b; \text{return } (f\ a'\ b') \}
\end{align*}

- Lifting allows to apply pure functions point-free to monadic values
- Together with monadic bind it constitutes a functional approach as opposed to the do-notation
**liftM**

**liftM and liftM2 definition**

\[\text{liftM} :: (\text{Monad } m) \Rightarrow (a \to b) \to (m a \to m b)\]

\[\text{liftM } f = \lambda a \to \text{do } \{ a' \leftarrow a; \text{return } (f \ a') \}\]

\[\text{liftM2} :: (\text{Monad } m) \Rightarrow
\quad (a \to b \to c) \to (m a \to m b \to m c)\]

\[\text{liftM2 } f = \quad
\lambda a \ b \to \text{do } \{ a' \leftarrow a; b' \leftarrow b; \text{return } (f \ a' \ b') \}\]

- Lifting allows to apply pure functions point-free to monadic values
- Together with monadic bind it constitutes a functional approach as opposed to the do-notation
**liftM example 1**

```
getName :: String → Maybe String
getName name = do
  let db =
      [ ("John","Smith, John"),
        ("Mike","Caine, Michael") ]
  tempName ← lookup name db
  return (swapNames tempName)

Can be rewritten as:

```
getName name = do
  let db = ...
  liftM swapNames (lookup name db)
```
liftM example 2

\[
\text{addDataFromFile} :: \text{Dict} \to \text{Handle} \to \text{IO Dict}
\]

\[
\text{addDataFromFile} \ \text{dict} \ \text{hdl} = \text{do}
\]
\[
\text{contents} \leftarrow \text{hGetContents hdl}
\]
\[
\text{entries} \leftarrow \text{return} \ (\text{map read} \ (\text{lines contents}))
\]
\[
\text{return} \ (\text{foldl} \ (\text{addEntry}) \ \text{dict} \ \text{entries})
\]

Can be rewritten as:

\[
\text{addDataFromFile} \ \text{dict} = \text{liftM} \ (\text{foldl} \ \text{addEntry} \ \text{dict} \ \circ \ \text{map read} \ \circ \ \text{lines}) \ \circ \ \text{hGetContents}
\]
liftM

**liftM2 example 1**

What does this do?

\[
\text{allCombinations} :: (a \rightarrow a \rightarrow a) \rightarrow [[a]] \rightarrow [a]
\]
\[
\text{allCombinations} \ fn \ [\ ] = [\ ]
\]
\[
\text{allCombinations} \ fn \ (l : ls) = \text{foldl} \ (\text{liftM2} \ fn) \ l \ ls
\]
**liftM**

**liftM2 example 1**

What does this do?

\[
\text{allCombinations} :: (a \rightarrow a \rightarrow a) \rightarrow [[a]] \rightarrow [a]
\]

\[
\text{allCombinations \ fn} \ [] = []
\]

\[
\text{allCombinations \ fn} \ (l : ls) = \text{foldl} (\text{liftM2 \ fn}) \ l \ ls
\]

**Output**

Main> \text{allCombinations} (+) \[[0,1],[1,2,3]\]
[0+1,0+2,0+3,1+1,1+2,1+3] = [1,2,3,2,3,4]

Main> \text{allCombinations} (*) \[[0,1],[1,2],[3,5]\]
[0+1,0+2,0+3,1+1,1+2,1+3] = [0,0,0,0,3,5,6,10]
**ap**

**ap definition**

`ap` helps when both function and argument are in the monad.

\[
ap :: (Monad m) \Rightarrow m (a \rightarrow b) \rightarrow m a \rightarrow m b
\]

\[
ap = \text{liftM2} (\$)
\]

Note that `liftM2 f x y` is equivalent to `return f \`ap\` x \`ap\` y.

**Output**

Main> `[(\*2),(+3)] \`ap\` [0,1,2]
[0,2,4,3,4,5]
Main> (Just \((\*2)) \`ap\` (Just 3)
Just 6
words :: String → [String]
lookup :: (Eq a) ⇒ a → [(a, b)] → Maybe b
ap :: (Monad m) ⇒ m (a → b) → m a → m b
main = do
  let fns = [("double", (2*)),"halve", ('div'2)),
            ("square", (λx → x * x)),"negate", negate),
            ("incr", (+1)),"decr", (+(-1))]
  args ← getArgs
  let val = read (args !! 0)
  cmds = map ((flip lookup) fns) (words (args !! 1))
  print $ foldl (flip ap) (Just val) cmds
Outline

1. List functions
2. Conditionals
3. Lifting
4. *MonadPlus* functions
**msum**

### msum definition

```haskell
class Monad m \Rightarrow MonadPlus m where
  mzero :: m a
  mplus :: m a \rightarrow m a \rightarrow m a
  msum :: MonadPlus m \Rightarrow [m a] \rightarrow m a
  msum xs = foldr mplus mzero xs
```
**msum example**

```haskell
type Variable = String
type Value = String
type EnvironmentStack = [[(Variable, Value)]]

lookupVar ::
    Variable → EnvironmentStack → Maybe Value
lookupVar var stack =
    msum $ map (lookup var) stack
```
**guard**

**guard definition**

\[
guard :: MonadPlus m \Rightarrow \text{Bool} \rightarrow m ()
guard p = \text{if } p \text{ then } \text{return } () \text{ else } \text{mzero}
\]

**guard example**

```haskell
data Record = Rec{name :: String, age :: Int}
type DB = [Record]

getYoungerThan :: Int \rightarrow DB \rightarrow [Record]
getYoungerThan limit db =
  mapMaybe (\r \rightarrow
    do {guard (age r < limit); return r}) db
```
List comprehensions

Syntax 1

list = [r | x1 <- xs1, x2 <- xs2, ..., b1, b2, ...]

Syntax 2

list = do x1 <- xs1
        x2 <- xs2
        ...
        guard b1
        guard b2
        ...
        return r