# Functional Programming

## Monadic Prelude

Jevgeni Kabanov

Department of Computer Science
University of Tartu

# Introduction

## Previously on Functional Programming

- Monadic laws
- *Monad* class ($\gg=$ and *return*)
- *MonadPlus* class (*mzero* and *mplus*)
- **do**-notation
- Maybe monad
- List monad
- State monad
- IO monad

# Overview

# 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 imperartive 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 imperartive 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 imperartive 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 imperartive languages
- In fact given the **do**-notation and these functions we can program in Haskell using imperative approach

# Outline

# Monadic Prelude

## *sequence* definition

$$
\begin{aligned}
&\mathit{sequence\_} :: \mathit{Monad}\ m \Rightarrow [m\ a] \rightarrow m\ ()\\
&\mathit{sequence\_} = \mathit{foldr}\ (\gg)\ (\mathit{return}\ ())\\
&\mathit{sequence} \quad :: \mathit{Monad}\ m \Rightarrow [m\ a] \rightarrow m\ [a]\\
&\mathit{sequence} \quad = \mathit{foldr}\ \mathit{mcons}\ (\mathit{return}\ [\,])\\
&\quad \textbf{where}\ \mathit{mcons}\ p\ q =\\
&\qquad p \gg\!= \lambda x \rightarrow q \gg\!= \lambda y \rightarrow \mathit{return}\ (x : y)
\end{aligned}
$$

# *sequence*

*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*

## *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 assymmetrical with respect to nothing!

# $mapM$

## $mapM$ definition

$$mapM :: Monad\ m \Rightarrow (a \to m\ b) \to [a] \to m\ [b]$$
$$mapM\ f\ as = sequence\ (map\ f\ as)$$

$$mapM\_ :: Monad\ m \Rightarrow (a \to m\ b) \to [a] \to m\ ()$$
$$mapM\_\ f\ as = sequence\_\ (map\ f\ as)$$

# $mapM$

## $mapM$ example 1

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

## $mapM$ example 2

$$putString :: [\,Char\,] \rightarrow IO\ ()$$
$$putString\ s = mapM\_\ putChar\ s$$

# $forM$

## $forM$ definition

$$
\begin{aligned}
&forM :: Monad\ m \Rightarrow [a] \to (a \to m\ b) \to m\ [b] \\
&forM = flip\ mapM \\
&forM\_ :: Monad\ m \Rightarrow [a] \to (a \to m\ b) \to m\ () \\
&forM\_ = flip\ mapM\_
\end{aligned}
$$

## $forM\_$ example

$$
\begin{aligned}
&main = \mathbf{do} \\
&\quad forM\_\ [1\,..\,10]\ (\lambda i \to print\ i)
\end{aligned}
$$

# *filterM*

## *filterM* definition

$$\begin{aligned}
&\mathit{filterM} :: \mathit{Monad}\ m \Rightarrow (a \rightarrow m\ \mathit{Bool}) \rightarrow [\,a\,] \rightarrow m\ [\,a\,]\\
&\mathit{filterM}\ \ p\ [\,]= \mathit{return}\ [\,]\\
&\mathit{filterM}\ \ p\ (x : xs) = \mathbf{do}\ b \leftarrow p\ x\\
&\quad ys \leftarrow \mathit{filterM}\ \ p\ xs\\
&\quad \mathit{return}\ (\mathbf{if}\ b\ \mathbf{then}\ (x : ys)\ \mathbf{else}\ ys)
\end{aligned}$$

## *filterM* example

$$\begin{aligned}
&\mathit{main} = \mathbf{do}\\
&\quad \mathit{names} \leftarrow \mathit{getArgs}\\
&\quad \mathit{dirs} \leftarrow \mathit{filterM}\ \ \mathit{doesDirectoryExist}\ \mathit{names}\\
&\quad \mathit{mapM_}\ \mathit{putStrLn}\ \mathit{dirs}
\end{aligned}$$

# *foldM*

## *foldM* definition

$$
\begin{aligned}
&foldl && :: (a \to b \to a) \to a \to [b] \to a \\
&foldl\ f\ z\ [\,] && = \quad z \\
&foldl\ f\ z\ (x : xs) && = foldl\ f\ (f\ z\ x)\ xs
\end{aligned}
$$

$$
\begin{aligned}
&foldM :: (Monad\ m) \Rightarrow (a \to b \to m\ a) \to a \to [b] \to m\ a \\
&foldM\ f\ a\ [\,] = return\ a \\
&foldM\ f\ a\ (x : xs) = f\ a\ x \gg= \lambda y \to foldM\ f\ y\ xs
\end{aligned}
$$

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

# *foldM*

## *foldM* explanation

$$
\begin{aligned}
&foldM\ f\ a1\ [x1, x2, ..., xn] = \mathbf{do}\\
&\quad a2 \leftarrow f\ a1\ x1\\
&\quad a3 \leftarrow f\ a2\ x2\\
&\quad \ldots\\
&\quad f\ an\ xn
\end{aligned}
$$

# *foldM*

## *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

$$
\begin{array}{l}
\textbf{data } \mathit{Sheep} = \mathit{Sheep}\{ \mathit{name} :: \mathit{String}, \\
\quad \mathit{mother} :: \mathit{Maybe\ Sheep}, \mathit{father} :: \mathit{Maybe\ Sheep} \} \\
\mathit{dolly} :: \mathit{Sheep} \\
\mathit{dolly} = \textbf{let} \\
\quad \mathit{adam} \;\; = \mathit{Sheep}\ \texttt{"Adam"}\ \mathit{Nothing}\ \mathit{Nothing} \\
\quad \mathit{eve} \;\;\;\; = \mathit{Sheep}\ \texttt{"Eve"}\ \mathit{Nothing}\ \mathit{Nothing} \\
\quad \mathit{uranus} = \mathit{Sheep}\ \texttt{"Uranus"}\ \mathit{Nothing}\ \mathit{Nothing} \\
\quad \mathit{gaea} \;\;\, = \mathit{Sheep}\ \texttt{"Gaea"}\ \mathit{Nothing}\ \mathit{Nothing} \\
\quad \mathit{kronos} = \mathit{Sheep}\ \texttt{"Kronos"}\ (\mathit{Just\ gaea})\ (\mathit{Just\ uranus}) \\
\quad \mathit{holly} \;\; = \mathit{Sheep}\ \texttt{"Holly"}\ (\mathit{Just\ eve})\ (\mathit{Just\ adam}) \\
\quad \mathit{roger} \;\; = \mathit{Sheep}\ \texttt{"Roger"}\ (\mathit{Just\ eve})\ (\mathit{Just\ kronos}) \\
\quad \mathit{molly} \;\; = \mathit{Sheep}\ \texttt{"Molly"}\ (\mathit{Just\ holly})\ (\mathit{Just\ roger}) \\
\textbf{in } \mathit{Sheep}\ \texttt{"Dolly"}\ (\mathit{Just\ molly})\ \mathit{Nothing}
\end{array}
$$

# *foldM*

## *foldM* example 2

$$
\begin{aligned}
&traceFamily ::\\
&\quad [(Sheep \rightarrow Maybe\ Sheep)] \rightarrow Sheep \rightarrow Maybe\ Sheep\\
&traceFamily\ l\ s = foldM\ (flip\ (\$))\ s\ l\\
&paternalGrandmother =\\
&\quad traceFamily\ [father, mother]\\
&mothersPaternalGrandfather =\\
&\quad traceFamily\ [mother, father, father]
\end{aligned}
$$

Output:

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

# *foldM*

## *Map* definition

$$\textbf{data}\ Map\ k\ a$$

$$\begin{aligned}
&empty :: Map\ k\ a \\
&insert :: Ord\ k \Rightarrow k \to a \to Map\ k\ a \to Map\ k\ a \\
&lookup :: (Monad\ m, Ord\ k) \Rightarrow k \to Map\ k\ a \to m\ a \\
&toList :: Map\ k\ a \to [(k, a)]
\end{aligned}$$

# *foldM*

## *foldM* example 3

$$\mathbf{data}\ Entry = Entry\{key :: String, value :: String\}$$
$$\mathbf{type}\ Dict = Map\ String\ String$$

$$addEntry :: Dict \to Entry \to Dict$$
$$addEntry\ d\ e = insert\ (key\ e)\ (value\ e)\ d$$

$$addDataFromFile :: Dict \to Handle \to IO\ Dict$$
$$addDataFromFile\ dict\ hdl = \mathbf{do}$$
$$\quad contents \leftarrow hGetContents\ hdl$$
$$\quad entries \leftarrow return\ (map\ read\ (lines\ contents))$$
$$\quad return\ (foldl\ (addEntry)\ dict\ entries)$$

# $foldM$

## $foldM$ example 3

$$
\begin{array}{l}
main :: IO\ ()\\
main = \mathbf{do}\\
\quad files \leftarrow getArgs\\
\quad handles \leftarrow mapM\ openForReading\ files\\
\quad dict \leftarrow foldM\ addDataFromFile\ empty\ handles\\
\quad print\ (toList\ dict)
\end{array}
$$

# *join*

## *join* definition

$$
\begin{aligned}
&join \quad :: (Monad\ m) \Rightarrow m\ (m\ a) \rightarrow m\ a \\
&join\ x = x \gg\!= id
\end{aligned}
$$

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

## *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

# *when*

## *when* and *unless* definition

$$when :: (Monad\ m) \Rightarrow Bool \rightarrow m\ () \rightarrow m\ ()$$
$$when\ p\ s = \textbf{if}\ p\ \textbf{then}\ s\ \textbf{else}\ return\ ()$$
$$unless :: (Monad\ m) \Rightarrow Bool \rightarrow m\ () \rightarrow m\ ()$$
$$unless\ p\ s = when\ (\neg\ p)\ s$$

## *when* example

```
Monads> mapM_ (\l -> when (not $ null l) (putStrLn l))
                      ["","abc","def","","","ghi"]
abc
def
ghi
```

# Outline

# *liftM*

## *liftM* and *liftM2* definition

$$liftM :: (Monad\ m) \Rightarrow (a \rightarrow b) \rightarrow (m\ a \rightarrow m\ b)$$
$$liftM\ f = \lambda a \rightarrow \mathbf{do}\ \{a' \leftarrow a; return\ (f\ a')\}$$
$$liftM2 :: (Monad\ m) \Rightarrow$$
$$\quad (a \rightarrow b \rightarrow c) \rightarrow (m\ a \rightarrow m\ b \rightarrow m\ c)$$
$$liftM2\ f =$$
$$\quad \lambda a\ b \rightarrow \mathbf{do}\ \{a' \leftarrow a; b' \leftarrow b; return\ (f\ a'\ b')\}$$

- 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

# *liftM*

## *liftM* and *liftM2* definition

$$
\begin{aligned}
&\mathit{liftM} :: (\mathit{Monad}\ m) \Rightarrow (a \rightarrow b) \rightarrow (m\ a \rightarrow m\ b)\\
&\mathit{liftM}\ f = \lambda a \rightarrow \mathbf{do}\ \{a' \leftarrow a; \mathit{return}\ (f\ a')\}\\
&\mathit{liftM2} :: (\mathit{Monad}\ m) \Rightarrow\\
&\quad (a \rightarrow b \rightarrow c) \rightarrow (m\ a \rightarrow m\ b \rightarrow m\ c)\\
&\mathit{liftM2}\ f =\\
&\quad \lambda a\ b \rightarrow \mathbf{do}\ \{a' \leftarrow a; b' \leftarrow b; \mathit{return}\ (f\ a'\ b')\}
\end{aligned}
$$

- 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

# *liftM*

## *liftM* and *liftM2* definition

$$liftM :: (Monad\ m) \Rightarrow (a \to b) \to (m\ a \to m\ b)$$
$$liftM\ f = \lambda a \to \mathbf{do}\ \{a' \leftarrow a; return\ (f\ a')\}$$
$$liftM2 :: (Monad\ m) \Rightarrow$$
$$(a \to b \to c) \to (m\ a \to m\ b \to m\ c)$$
$$liftM2\ f =$$
$$\lambda a\ b \to \mathbf{do}\ \{a' \leftarrow a; b' \leftarrow b; return\ (f\ a'\ b')\}$$

- 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

# *liftM*

## *liftM* example 1

$$
\begin{array}{l}
getName :: String \to Maybe\ String \\
getName\ name = \textbf{do} \\
\quad \text{let } db = \\
\qquad [(\texttt{"John"}, \texttt{"Smith, John"}), \\
\qquad (\texttt{"Mike"}, \texttt{"Caine, Michael"})] \\
\quad tempName \leftarrow lookup\ name\ db \\
\quad return\ (swapNames\ tempName)
\end{array}
$$

Can be rewritten as:

$$
\begin{array}{l}
getName\ name = \textbf{do} \\
\quad \text{let } db = \ldots \\
\quad liftM\ swapNames\ (lookup\ name\ db)
\end{array}
$$

# *liftM*

## *liftM* example 2

$$
\begin{aligned}
&\mathit{addDataFromFile} :: \mathit{Dict} \rightarrow \mathit{Handle} \rightarrow \mathit{IO}\ \mathit{Dict}\\
&\mathit{addDataFromFile}\ \mathit{dict}\ \mathit{hdl} = \mathbf{do}\\
&\quad \mathit{contents} \leftarrow \mathit{hGetContents}\ \mathit{hdl}\\
&\quad \mathit{entries} \leftarrow \mathit{return}\ (\mathit{map}\ \mathit{read}\ (\mathit{lines}\ \mathit{contents}))\\
&\quad \mathit{return}\ (\mathit{foldl}\ (\mathit{addEntry})\ \mathit{dict}\ \mathit{entries})
\end{aligned}
$$

Can be rewritten as:

$$
\begin{aligned}
&\mathit{addDataFromFile}\ \mathit{dict} =\\
&\quad \mathit{liftM}\ (\mathit{foldl}\ \mathit{addEntry}\ \mathit{dict} \circ \mathit{map}\ \mathit{read} \circ \mathit{lines})\\
&\qquad \circ\, \mathit{hGetContents}
\end{aligned}
$$

# *liftM*

## *liftM2* example 1

What does this do?

$$allCombinations :: (a \to a \to a) \to [[a]] \to [a]$$
$$allCombinations\ fn\ [\,] = [\,]$$
$$allCombinations\ fn\ (l : ls) = foldl\ (liftM2\ fn)\ l\ ls$$

# $liftM$

## $liftM2$ example 1

What does this do?

$$
\begin{aligned}
&allCombinations :: (a \to a \to a) \to [[a]] \to [a] \\
&allCombinations\ fn\ [\,] = [\,] \\
&allCombinations\ fn\ (l : ls) = foldl\ (liftM2\ fn)\ l\ ls
\end{aligned}
$$

## Output

```
Main> 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> 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.

$$ap :: (Monad\ m) \Rightarrow m\ (a \rightarrow b) \rightarrow m\ a \rightarrow m\ b$$
$$ap = liftM2\ (\$)$$

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
```

# $ap$

## $ap$ example

$$words :: String \to [String]$$
$$lookup :: (Eq\ a) \Rightarrow a \to [(a, b)] \to Maybe\ b$$
$$ap :: (Monad\ m) \Rightarrow m\ (a \to b) \to m\ a \to m\ b$$

$main = \mathbf{do}$
  $\text{let } fns =$
    $[(\texttt{"double"}, (2*)), (\texttt{"halve"}, (`div`2)),$
    $(\texttt{"square"}, (\lambda x \to x * x)), (\texttt{"negate"}, negate),$
    $(\texttt{"incr"}, (+1)), (\texttt{"decr"}, (+(-1)))]$
  $args \leftarrow getArgs$
  $\text{let } val = read\ (args\ !!\ 0)$
    $cmds = map\ ((flip\ lookup)\ fns)\ (words\ (args\ !!\ 1))$
  $print\ \$\ foldl\ (flip\ ap)\ (Just\ val)\ cmds$

# Outline

# *msum*

## *msum* definition

$$\mathbf{class}\ Monad\ m \Rightarrow MonadPlus\ m\ \mathbf{where}$$
$$\quad mzero :: m\ a$$
$$\quad mplus :: m\ a \to m\ a \to m\ a$$
$$msum :: MonadPlus\ m \Rightarrow [m\ a] \to m\ a$$
$$msum\ xs = foldr\ mplus\ mzero\ xs$$

# *msum*

## *msum* example

$$\mathbf{type}\ Variable = String$$
$$\mathbf{type}\ Value = String$$
$$\mathbf{type}\ EnvironmentStack = [[(Variable, Value)]]$$

$$\begin{aligned}&lookupVar :: \\ &\quad Variable \to EnvironmentStack \to Maybe\ Value \\ &lookupVar\ var\ stack = \\ &\quad msum \ \$\ map\ (lookup\ var)\ stack\end{aligned}$$

# *guard*

## *guard* definition

$$
\begin{aligned}
&\mathit{guard} :: \mathit{MonadPlus}\ m \Rightarrow \mathit{Bool} \rightarrow m\ ()\\
&\mathit{guard}\ p = \mathbf{if}\ p\ \mathbf{then}\ \mathit{return}\ ()\ \mathbf{else}\ \mathit{mzero}
\end{aligned}
$$

## *guard* example

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Record} = \mathit{Rec}\{\mathit{name} :: \mathit{String}, \mathit{age} :: \mathit{Int}\}\\
&\mathbf{type}\ \mathit{DB} = [\mathit{Record}]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{getYoungerThan} :: \mathit{Int} \rightarrow \mathit{DB} \rightarrow [\mathit{Record}]\\
&\mathit{getYoungerThan}\ \mathit{limit}\ \mathit{db} =\\
&\quad \mathit{mapMaybe}\ (\lambda r \rightarrow\\
&\qquad \mathbf{do}\ \{\mathit{guard}\ (\mathit{age}\ r < \mathit{limit}); \mathit{return}\ r\})\ \mathit{db}
\end{aligned}
$$

# 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
```