301AA - Advanced Programming
[AP-2017]

Lecturer: Andrea Corradini andrea@di.unipi.it
Tutor: Lillo Galletta galletta@di.unipi.it

Department of Computer Science, Pisa
Academic Year 2017/18

AP-2017-21: Monads in Haskell
Pros of Functional Programming

- Functional programming is beautiful:
  - Concise and powerful abstractions
    - higher-order functions, algebraic data types, parametric polymorphism, principled overloading, ...
  - Close correspondence with mathematics
    - Semantics of a code function is the mathematical function
    - Equational reasoning: if $x = y$, then $f\ x = f\ y$
    - Independence of order-of-evaluation (Confluence, aka Church-Rosser)

The compiler can choose the best sequential or parallel evaluation order
Problems...

• But to be *useful*, a language must be able to manage “impure features”:
  – Input/Output
  – Imperative update
  – Error recovery (eg, timeout, divide by zero, etc.)
  – Foreign-language interfaces
  – Concurrency control

The whole point of running a program is to interact with the external environment and affect it
The Direct Approach

• Just add imperative constructs “the usual way”
  – I/O via “functions” with side effects:
    
    ```
    putchar 'x' + putchar 'y'
    ```
  – Imperative operations via assignable reference cells:
    ```
    z = ref 0; z := !z + 1;
    f(z);
    w = !z  (* What is the value of w? *)
    ```
  – Error recovery via exceptions
  – Foreign language procedures mapped to “functions”
  – Concurrency via operating system threads
• Can work *if language determines evaluation order*
  – Ocaml, Standard ML are good examples of this approach
But what if we are “lazy”?

In a lazy functional language, like Haskell, the order of evaluation is undefined.

• Example: \[ \text{res} = \text{putchar 'x'} + \text{putchar 'y'} \]
  – Output depends upon the evaluation order of (+).

• Example: \[ \text{ls} = [\text{putchar 'x'}, \text{putchar 'y'}] \]
  – Output depends on how list is used
  – If only used in \text{length ls}, nothing will be printed because \text{length} does not evaluate elements of list
Fundamental question

• Is it possible to add imperative features without changing the meaning of pure Haskell expressions?

• **Yes!** Using the concept of **monad**
  – Formally defined as a **type constructor class**
  – Each **monadic type constructor** defines certain **monadic values** (sometimes called **actions**) and how to **compose them sequentially**
Type Constructor Classes
Type Constructor Classes

• **Type Classes** are predicates over **types**
• **[Type] Constructor Classes** are predicates over **type constructors**
• Allow to define overloaded functions common to several type constructors
• Example: **map** function useful on many Haskell types
  - Lists:

```
map:: (a -> b) -> [a] -> [b]
map f  [] = []
map f (x:xs) = f x : map f xs

> map (\x->x+1) [1,2,4]
[2,3,5]
```
More examples of \texttt{map} function

\begin{verbatim}
data Tree a = Leaf a | Node(Tree a, Tree a)
deriving Show

mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f (Leaf x) = Leaf (f x)
mapTree f (Node(l,r)) = Node (mapTree f l, mapTree f r)

> t1 = Node(Node(Leaf 3, Leaf 4), Leaf 5)
> mapTree (\x->x+1) t1
Node (Node (Leaf 4,Leaf 5),Leaf 6)
\end{verbatim}

\begin{verbatim}
data Maybe a = Nothing | Just a
deriving Show

mapMaybe :: (a -> b) -> Maybe a -> Maybe b
mapMaybe f Nothing = Nothing
mapMaybe f (Just x) = Just (f x)

> o1 = Just 10
> mapMaybe (\x->x+1) o1
Just 11
\end{verbatim}
Constructor Classes

- All map functions share the same structure:

  ```haskell
  map :: (a -> b) -> [a] -> [b]
  mapTree :: (a -> b) -> Tree a -> Tree b
  mapMaybe :: (a -> b) -> Maybe a -> Maybe b
  ```

- They can all be written as:

  ```haskell
  fmap :: (a -> b) -> g a -> g b
  ```

  — where $g$ is:

  - [-] for lists, `Tree` for trees, and `Maybe` for options

- Note that $g$ is a function from types to types, i.e. a **type constructor**
Constructor Classes

• This pattern can be captured in a constructor class **Functor**:

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

• Simply a **type class** where the predicate is over a **type constructors** rather than on a **type**

• Compare with the definition of a standard type class:

```haskell
class Eq a where
  (==) :: a -> a -> Bool
```
The **Functor** constructor class
and some instances

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

instance Functor [] where
    fmap f [] = []
    fmap f (x:xs) = f x : fmap f xs

instance Functor Tree where
    fmap f (Leaf x) = Leaf (f x)
    fmap f (Node(t1,t2)) = Node(fmap f t1, fmap f t2)

instance Functor Maybe where
    fmap f (Just s) = Just(f s)
    fmap f Nothing = Nothing
The **Functor** constructor class and some instances (2)

- Or by reusing the definitions map, mapTree, and mapMaybe:

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

instance Functor [] where
    fmap = map

instance Functor Tree where
    fmap = mapTree

instance Functor Maybe where
    fmap = mapMaybe
```
Constructor Classes

• We can then use the **overloaded** symbol `fmap` to map over all three kinds of data structures:

```haskell
*Main> fmap (\x->x+1) [1,2,3]
[2,3,4]
it :: [Integer]

*Main> fmap (\x->x+1) (Node(Leaf 1, Leaf 2))
Node (Leaf 2,Leaf 3)
it :: Tree Integer

*Main> fmap (\x->x+1) (Just 1)
Just 2
it :: Maybe Integer
```

• The **Functor** constructor class is part of the standard Prelude for Haskell
Towards **Monads**

- Often type constructors can be thought of as defining "boxes" for values
- **Functors** with `fmap` allow to apply functions inside "boxes"
- **Monads** are constructor classes introducing operations for
  - Putting a value into a "box" (**return**)
  - Compose functions that return "boxed" values (**bind**)
The **Maybe** type constructor

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

-- example
```haskell```
```text```
sqrt :: Int -> Maybe Real

• A value of type **Maybe a** is a possibly undefined value of type **a**

• A function **f :: a -> Maybe b** is a **partial function** from **a** to **b**
Composing partial function

father :: Person -> Maybe Person  -- partial function
mother :: Person -> Maybe Person  -- (lookup in a DB)

maternalGrandfather :: Person -> Maybe Person
maternalGrandfather p =
    case mother p of
        Nothing -> Nothing
        Just mom -> father mom

bothGrandfathers :: Person -> Maybe (Person, Person)
bothGrandfathers p =
    case father p of
        Nothing -> Nothing
        Just dad ->
            case father dad of
                Nothing -> Nothing
                Just gf1 ->
                    case mother p of
                        Nothing -> Nothing
                        Just mom ->
                            case father mom of
                                Nothing -> Nothing
                                Just gf2 ->
                                    Just (gf1, gf2)

-- found first grandfather
-- found second grandfather
The **Monad** type class and the **Maybe** monad

```haskell
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b -- "bind"
    ... -- + something more
```

- **m** is a type constructor
- **m a** is the type of **monadic values**

```haskell
instance Monad Maybe where
    return :: a -> Maybe a
    return x = Just x

    (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
    y >>= g = case y of
        Nothing -> Nothing
        Just x -> g x
```

- **bind (>>=)** shows how to “propagate” undefinedness
Use of **bind** of the **Maybe** monad to compose partial functions

```
father :: Person -> Maybe Person  -- partial function
mother :: Person -> Maybe Person  -- (lookup in a DB)

maternalGrandfather :: Person -> Maybe Person
maternalGrandfather p =
    case mother p of
      Nothing -> Nothing
      Just mom -> father mom

maternalGrandfather p = mother p >>= father

bothGrandfathers :: Person -> Maybe(Person, Person)
bothGrandfathers p =
    father p >>=
        (
dad -> father dad >>=
            (gf1 -> mother p >>=
                (mom -> father mom >>=
                    (gf2 -> return (gf1,gf2) ))))))
```
Alternative, imperative-style syntax: \textbf{do}

\begin{verbatim}
bothGrandfathers p =
  father p >>=
    (\dad -> father dad >>=
      (\gf1 -> mother p >>=
        (\mom -> father mom >>=
          (\gf2 -> return (gf1, gf2) ))))
\end{verbatim}

\begin{verbatim}
bothGrandfathers p = do {
  dad <- father p;
  gf1 <- father dad;
  mom <- mother p;
  gf2 <- father mom;
  return (gf1, gf2);
}
\end{verbatim}

- \textbf{do} syntax is just syntactic sugar for \texttt{>>=}

20
Some Haskell Monads

<table>
<thead>
<tr>
<th>Monad</th>
<th>Imperative semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maybe</td>
<td>Exception (Anonymous)</td>
</tr>
<tr>
<td>Error</td>
<td>Exception (with error description)</td>
</tr>
<tr>
<td>State</td>
<td>Global state</td>
</tr>
<tr>
<td>IO</td>
<td>Input/output</td>
</tr>
<tr>
<td>[] (lists)</td>
<td>Non-determinism</td>
</tr>
<tr>
<td>Reader</td>
<td>Environment</td>
</tr>
<tr>
<td>Writer</td>
<td>Logger</td>
</tr>
</tbody>
</table>
Understanding Monads as containers

class Monad m where -- definition of Monad type class
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b -- "bind"
    ...

• The monadic constructor can be seen as a **container**: let’s see this for **lists**

• Getting **bind** from more basic operations

```haskell
map :: (a -> b) -> [a] -> [b] -- seen. "fmap" for Functors

return :: a -> [a] -- container with single element
return x = [x]

concat :: [[a]] -> [a] -- flattens two-level containers

Example: concat [[1,2],[[],[4]]] = [1,2,4]
```

```haskell
(>>=) :: [a] -> (a -> [b]) -> [b]
xs >>= f = concat(map f xs)
```

Exercise: define **map** and **concat** using **bind** and **return**
Understanding Monads as computations

A value of type \( m \ a \) is a computation returning a value of type \( a \).

For any value, there is a computation which “does nothing” and produces that result. This is given by function \( \text{return} \).

Given two computations \( x \) and \( y \), one can form the computation \( x \gg y \) which intuitively “runs” \( x \), throws away its result, then runs \( y \) returning its result.

Given computation \( x \), we can use its result to decide what to do next. Given \( f : a \to m \ b \), computation \( x \gg= f \) runs \( x \), then applies \( f \) to its result, and runs the resulting computation.

Note that we can define then using bind:
\[
x \gg y = x \gg= (\_ \to y)
\]
Understanding Monads as computations (2)

```haskell
class Monad m where -- definition of Monad type class
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b -- "bind"
    (>>)  :: m a -> m b -> m b -- "then"
    ...
```

- `return`, `bind` and `then` define basic ways to compose computations
- They are used in Haskell libraries to define more complex composition operators and control structures (sequence, for-each loops, ...)
- If a type constructor defining a library of computations is `monadic`, one gets automatically benefit of such libraries

**Example:** MAYBE

- \( f : a \rightarrow \text{Maybe } b \) is a `partial` function
- `bind` applies a partial function to a possibly undefined value, propagating undefinedness

**Example:** LISTS

- \( f : a \rightarrow [b] \) is a `non-deterministic` function
- `bind` applies a non-deterministic function to a list of values, collecting all possible results

**Example:** Parsing, handling errors, IO, backtracking....
Problem

Monadic Input and Output
The IO Monad

A functional program defines a pure function, with no side effects

The whole point of running a program is to have some side effect

The term “side effect” itself is misleading
Before Monads

• Streams
  – Program sends stream of requests to OS, receives stream of responses

• Continuations
  – User supplies continuations to I/O routines to specify how to process results

• Haskell 1.0 Report adopted Stream model
  – Stream and Continuation models were discovered to be inter-definable
Stream Model: Basic Idea

- Move “side effects” outside of functional program
- Haskell `main :: String -> String`

But what if you need to read more than one file? Or delete files? Or communicate over a socket? ...
Stream Model

• Enrich argument and return type of `main` to include all input and output events.

```haskell
main :: [Response] -> [Request]
data Request = ReadFile Filename
   | WriteFile FileName String
   | ...
data Response = RequestFailed
   | ReadOK String
   | WriteOk
   | Success   | ...
```

• Wrapper program interprets requests and adds responses to input.

• Move side effects outside of functional program
Stream Model: **main :: [Response] -> [Request]**

- Problem: Laziness allows program to generate requests prior to processing any responses.
- Hard to extend
  - New I/O operations require adding new constructors to Request and Response types, modifying wrapper
- Does not associate Request with Response
  - easy to get “out-of-step,” which can lead to deadlock
- Not composable
  - no easy way to combine two “main” programs
- ... and other problems!!!
Monadic I/O: The Key Ideas

• *IO* is a type constructor, instance of *Monad*
• A value of type *(IO t)* is a computation or “action” that, when performed, may do some input/output before delivering a result of type *t*
• *return* returns the value without making I/O
• *then* (>>>) [and also *bind* (>>=)] composes two actions sequentially into a larger action
• The only way to perform an action is to call it at some point, directly or indirectly, from *Main.main*
A value of type \((\text{IO } t)\) is an “action.” When performed, it may do some input/output before delivering a result of type \(t\).

\[
\text{type } \text{IO } t = \text{World} \rightarrow (t, \text{World})
\]

- An action is a first-class value
- **Evaluating** an action has no effect; **performing** the action has the effect
Simple I/O actions

getChar :: IO Char
putChar :: Char -> IO ()

main :: IO ()
main = putChar 'x'

Main program is an action of type IO ()
The Bind Combinator ( >>= )

\[
(\gg\gg=) : : IO a \to (a \to IO b) \to IO b
\]

- We have connected two actions to make a new, bigger action.

```
getChar :: IO Char
putChar :: Char -> IO ()

getChar >>= putChar
```

```
echo :: IO ()
echo = getChar >>= putChar
```
The ( >>= ) Combinator

( >>= ) :: IO a -> (a -> IO b) -> IO b

• Operator is called bind because it binds the result of the left-hand action in the action on the right

• Performing compound action \(a \gg= \ \lambda x\to b:\)
  – performs action \(a\), to yield value \(r\)
  – applies function \(\lambda x\to b\) to \(r\)
  – performs the resulting action \(b\{x <- r\}\)
  – returns the resulting value \(v\)
The (>>) Combinator

• The “then” combinator (>>) does sequencing when there is no value to pass:

\[
(\gg) :: \text{IO } a \rightarrow \text{IO } b \rightarrow \text{IO } b
\]

\[
\text{defined from } \text{bind}
\]

\[
(\gg\gg) :: \text{IO } a \rightarrow (a \rightarrow \text{IO } b) \rightarrow \text{IO } b
\]

\[
m \gg n = m \gg\gg (\_ \rightarrow n)
\]

echoDup :: IO ()
echoDup = getChar \gg\gg \_ \rightarrow putChar c >> putChar c

echoTwice :: IO ()
echoTwice = echo >> echo
The return Combinator

- The action (return v) does no IO and immediately returns v:

```haskell
return :: a -> IO a
```

```haskell
getTwoChars :: IO (Char,Char)
getTwoChars = getChar >>= \c1 ->
             getChar >>= \c2 ->
             return (c1,c2)
```
The “do” Notation

• The “do” notation adds syntactic sugar to make monadic code easier to read.

```haskell
-- Plain Syntax
getTwoChars :: IO (Char,Char)
getTwoChars = getChar >>= \c1 ->
      getChar >>= \c2 ->
      return (c1,c2)
```

```haskell
-- Do Notation
getTwoCharsDo :: IO(Char,Char)
getTwoCharsDo = do { c1 <- getChar ;
                  c2 <- getChar ;
                  return (c1,c2) }
```

• **do** syntax designed to look imperative.
Desugaring “do” Notation

• The “do” notation only adds syntactic sugar:

```haskell
do { x }     = x
do { x; stmts }  = x >>= do { stmts }
do { v<-x; stmts } = x >>= \v -> do { stmts }
do {let ds; stmts } = let ds in do { stmts }
```

The scope of variables bound in a generator is the rest of the “do” expression.

• The following are equivalent:

```haskell
do { x1 <- p1; ...; xn <- pn; q }
```
Bigger Example

• The \texttt{getLine} function reads a line of input:

\begin{verbatim}
getLine :: IO [Char]
getLine = do { c <- getChar ;
  if c == '\n' then
    return []
  else
    do { cs <- getLine;
         return (c:cs) }
\end{verbatim}

Note the “regular” code mixed with the monadic operations and the nested “do” expression.
Control Structures on Monads

- Exploiting the monadic combinators, we can define control structures that work for any monad

```haskell
repeatN 0 x = return ()
repeatN n x = x >>= repeatN (n-1) x
repeatN :: (Num a, Monad m, Eq a) => a -> m a1 -> m ()
```

Main> repeatN 5 (putChar 'h')

```haskell
for [] fa = return ()
for (x:xs) fa = fa x >>= for xs fa
for :: Monad m => [t] -> (t -> m a) -> m ()
```

Main> for [1..10] (\x -> putStr (show x))
Sequencing

• Example use:

```haskell
Main> sequence [getChar, getChar, getChar]
```

```haskell
sequence :: [IO a] -> IO [a]
sequence [] = return []
sequence (a:as) = do { r <- a;
                     rs <- sequence as;
                     return (r:rs) }
sequence :: Monad m => [m a] -> m [a]
```
IO Provides Access to Files

• The IO Monad provides a large collection of operations for interacting with the “World.”

• For example, it provides a direct analogy to the Standard C library functions for files:

```haskell
openFile :: FilePath -> IOMode -> IO Handle
hPutStr   :: Handle    -> String   -> IO ()
hGetLine  :: Handle    -> IO String
hClose    :: Handle    -> IO ()
```
The IO operations let us write programs that do I/O in a strictly sequential, imperative fashion.

Idea: We can leverage the sequential nature of the IO monad to do other imperative things.

- A value of type `IORef a` is a reference to a mutable cell holding a value of type `a`.

```haskell
data IORef a  -- Abstract type
newIORef :: a -> IO (IORef a)
readIORef :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
```
Example Using References

```haskell
import Data.IORef -- import reference functions

-- Compute the sum of the first n integers
count :: Int -> IO Int
count n = do
  { r <- newIORef 0;
    addToN r 1 }

  where
    addToN :: IORef Int -> Int -> IO Int
    addToN r i | i > n     = readIORef r
                 | otherwise = do
      { v <- readIORef r
        ; writeIORef r (v + i)
        ; addToN r (i+1)}
```

This is terrible: Contrast with: sum [1..n].
The IO Monad as ADT

```haskell
return :: a -> IO a
(>>=) :: IO a -> (a -> IO b) -> IO b

getChar :: IO Char
putChar :: Char -> IO ()
... more operations on characters ...
openFile :: [Char] -> IOMode -> IO Handle
... more operations on files ...
newIORef :: a -> IO (IORef a)
... more operations on references ...
```

- All operations return an IO action, but only `bind (>>=)` takes one as an argument.
- **Bind** is the only operation that combines IO actions, which forces sequentiality.
- In **pure Haskell**, there is no way to transform a value of type `IO a` into a value of type `a`
Implementation of the IO monad

- GHC uses “world-passing semantics” for the IO monad

\[
\text{type IO } t = \text{World } \rightarrow (t, \text{World})
\]

- It represents the “world” by an un-forgeable token of type \(\text{World}\), and implements \text{bind} and \text{return} as:

\[
\begin{align*}
\text{return} :: a & \rightarrow \text{IO } a \\
\text{return } a & = \lambda w \rightarrow (a, w) \\
(\gg\gg=) :: \text{IO } a & \rightarrow (a \rightarrow \text{IO } b) \rightarrow \text{IO } b \\
(\gg\gg=) m k & = \lambda w \rightarrow \text{case } m w \text{ of } (r,w') \rightarrow k r w'
\end{align*}
\]

- Using this form, the compiler can do its normal optimizations. The dependence on the world ensures the resulting code will still be single-threaded.

- The code generator then converts the code to modify the world “in-place.”
Unreasonable Restriction?

- In **pure Haskell**, there is no way to transform a value of type `IO a` into a value of type `a`.
- Suppose you wanted to read a configuration file at the beginning of your program:
  ```haskell
  configFileContents :: [String]
  configFileContents = lines (readFile "config") -- WRONG!
  useOptimisation :: Bool
  useOptimisation = "optimise" `elem` configFileContents
  ```
- The problem is that `readFile` returns an `IO String`, not a `String`.
- **Option 1**: Write entire program in `IO monad`. But then we lose the simplicity of pure code.
- **Option 2**: Escape from the `IO Monad` using a function from `IO String -> String`. But this is disallowed!
Type-Unsafe Haskell Programming

• Reading a file is an I/O action, so in general it matters when we read the file.
• But we know the configuration file will not change during the program, so it doesn’t matter when we read it.
• This situation arises sufficiently often that Haskell implementations offer one last unsafe I/O primitive: `unsafePerformIO`.

```haskell
unsafePerformIO :: IO a -> a
configFileContents :: [String]
configFileContents = lines(unsafePerformIO(readFile "config"))
```
• The operator has a deliberately long name to discourage its use.

• Its use comes with a **proof obligation**: a promise to the compiler that the timing of this operation relative to all other operations doesn’t matter.
Warning: As its name suggests, \texttt{unsafePerformIO} breaks the soundness of the type system.

\begin{verbatim}
\texttt{r = unsafePerformIO (newIORef (error "urk"))}
\texttt{r :: IORef a \hspace{1cm} -- Type of the stored value is generic}

\texttt{cast x = unsafePerformIO (do {writeIORef r x; readIORef r})}
\texttt{\hspace{1cm} \geq \hspace{1cm} t (\x \rightarrow \texttt{cast x})}
\texttt{(\x \rightarrow \texttt{cast x}) :: a1 \rightarrow a2}
\texttt{\hspace{1cm} \geq \hspace{1cm} \texttt{cast 65:: Char}}
\texttt{\hspace{1cm} \geq \hspace{1cm} 'A'}
\end{verbatim}

So claims that Haskell is type safe only apply to programs that don’t use \texttt{unsafePerformIO}.

Similar examples are what caused difficulties in integrating references with Hindley/Milner type inference in ML.
Summary on Mondas

• A complete Haskell program is a single IO action called `main`. Inside IO, code is single-threaded.

• Big IO actions are built by gluing together smaller ones with `bind (>>=)` and by converting pure code into actions with return.

• IO actions are first-class.
  – They can be passed to functions, returned from functions, and stored in data structures.
  – So it is easy to define new “glue” combinators.

• The IO Monad allows Haskell to be pure while efficiently supporting side effects.

• The type system separates the pure from the effectful code.
Comparison

• In languages like ML or Java, the fact that the language is in the IO monad is baked in to the language. There is no need to mark anything in the type system because it is everywhere.

• In Haskell, the programmer can choose when to live in the IO monad and when to live in the realm of pure functional programming.

• So it is not Haskell that lacks imperative features, but rather the other languages that lack the ability to have a statically distinguishable pure subset.
Appendix: Monad Laws

1) \( \text{return } x \gg= f = f x \)
2) \( m \gg= \text{return } = m \)
3) \( (x \gg= f) \gg= g = x \gg= (\lambda v \rightarrow f v \gg= g) \)

• In do-notation:

1) \( \text{do } \{ w \leftarrow \text{return } v; f w \} \)
   \[ = \text{do } \{ f v \} \]

2) \( \text{do } \{ v \leftarrow x; \text{return } v \} \)
   \[ = \text{do } \{ x \} \]

3) \( \text{do } \{ x \leftarrow m1; y \leftarrow m2; m3 \} \)
   \[ = \text{do } \{ y \leftarrow \text{do } \{ x \leftarrow m1; m2 \}; m3 \} \]

\( x \) not in free vars of \( m3 \)
Derived Laws for (>>) and done

\[(\gg\gg) : \text{IO } a \to \text{IO } b \to \text{IO } b\]
\[m \gg n = m \gg\gg \_ \to n\]

\[\text{done} : \text{IO } ()\]
\[\text{done} = \text{return } ()\]

\[\text{done} \gg m = m\]
\[m \gg \text{done} = m\]
\[m_1 \gg (m_2 \gg m_3) = (m_1 \gg m_2) \gg m_3\]
Monads in Java: Optional and Stream

public static <T> Optional<T> of(T value)
// Returns an Optional with the specified present non-null value.

<U> Optional<U> flatMap(Function<? super T,Optional<U>> mapper)
/* If a value is present, apply the provided Optional-bearing mapping function to it, return that result, otherwise return an empty Optional. */

static <T> Stream<T> of(T t)
// Returns a sequential Stream containing a single element.

<R> Stream<R> flatMap(
    Function<? super T,? extends Stream<? extends R>> mapper)
/* Returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. */
Functional programming and monads in Java

• About the way monads entered the Java landscape I suggest reading the slides on Monadic Java by Mario Fusco.

• More on functional programming in Java in the book Java 8 in action