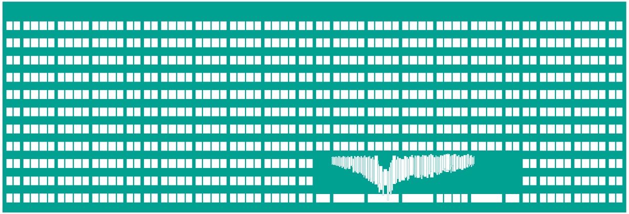
VŠB TECHNICKÁ UNIVERZITA OSTRAVA

VSB TECHNICAL UNIVERSITY OF OSTRAVA



www.vsb.cz

Monads in Haskell

behalek.cs.vsb.cz/wiki/Practical_Functional_Programming

Marek Běhálek

VSB – Technical University of Ostrava

marek.behalek@vsb.cz

November 4, 2022

VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Functional programming I

- Declarative style of programming
  - We define what needs to be computed, a run-time environment responsibility is how it will be evaluated.
  - Similar to math, we have various rules how to simplify an expression, but there are different ways how these rules can be applied for given expression.
- Programming with expressions (no statements)
  - Functional program is a set of function's definitions.
  - Functions are first class citizens - a function can return a function, high-order functions, partially evaluated functions.
  - Program's evaluation is the evaluation of some `main` expression.
- Immutable data structures - once created data can not be changed.
  - Studied problem, plenty of possibilities.
  - Common in API of many languages (C#: string, DateTime, `https://www.nuget.org/packages/System.Collections.Immutable/`).
  - Sometimes they are called persistent data structures.

Functional programming II

    - https://en.wikipedia.org/wiki/Persistent_data_structure
    - https://en.wikipedia.org/wiki/Persistent_array
  - What if I really need mutable data structure?
    - For example *quick* implementation of `quicksort`?
- No side effects
  - Functions only return values, no changes other changes.
  - For the same parameters, we always get the same result (referential transparency).
  - But, sometimes side effects can not be avoided (input - output operations) - how to solve that?

Functions with No Side Effects (1)

- What are side effects, how do i recognise them?

```
public double Add(double a, double b) {
    return a + b;
}
public double Add2(double a, double b) {
    try {
        Console.WriteLine($"a={a}, b={b}");
    } catch (Exception ex) { }
    return a + b;
}
public int Divide(int a, int b) {
    return a / b;
}
```

Functions with No Side Effects (2)

- How can I avoid them?

```
public int? Divide2(int a, int b) {
    if (b == 0)
        return null;
    return a / b;
}

public int Divide3(int a, NonZeroInteger b) {
    return a / b.Number;
}
```

```
public class NonZeroInteger {
    public int Number { get; }

    public NonZeroInteger(int number) {
        Number = number;
        if (number == 0)
            throw new ArgumentException();
    }
}
```

Expressions (1)

- Lets start with data type Maybe

```
data Maybe a = Nothing | Just a

betterDiv :: Int -> Int -> Maybe Int
betterDiv x y | y==0 = Nothing
              | otherwise = Just (x `div` y)
```

- Now we want to compute some *expressions* where we use it like a value type.

```
data Expr = Num Int
          | Add Expr Expr
          | Sub Expr Expr
          | Mul Expr Expr
          | Div Expr Expr
```

Expressions (2)

- Now we need to compute such expression

```
eval :: Expr -> Maybe Int
eval (Num x) = Just x
eval (Div x y) = case eval x of
                  Nothing -> Nothing
                  Just x' -> case eval y of
                      Nothing -> Nothing
                      Just y' -> betterDiv x' y'
eval (Add x y) = case eval x of
                  Nothing -> Nothing
                  Just x' -> case eval y of
                      Nothing -> Nothing
                      Just y' -> Just (x' + y')
```

- We can see emerging *patter*, how *actions* are *linked* one after the other.

Logging (1)

- Lets have simple *operations*.

```haskell
compute :: Int -> Int
compute x = x * x

isItEnough :: Int -> Bool
isItEnough x = x > 9
```

- We want then to compute values and *log* the context.

```haskell
compute :: (Int, String) -> (Int, String)
compute (x, log) = (x * x, log ++ "Just square of x.")

isItEnough :: (Int, String) -> (Bool, String)
isItEnough (x, log) = (x > 9, log ++ "Compared to 9.")
```

- It this a good solution? How to improve the quality of our solution?

Logging (2)

- What if I want to add the new entry at the start of log?

```
compute :: Int -> (Int, String)
compute x = (x * x, "Just square of x.")

isItEnough :: Int -> (Bool, String)
isItEnough x = (x > 9, "Compared to 9.")

applyLog :: (a,String) -> (a -> (b,String)) -> (b,String)
applyLog (x,log) f = let (y,newLog) = f x in (y,log ++ newLog)
```

```
*Main> applyLog (applyLog (2,"Initial value 2.") compute) isItEnough
(False,"Initial value 2.Just square of x.Compared to 9.")

*Main> (2,"Initial value 2.") `applyLog` compute `applyLog` isItEnough
(False,"Initial value 2.Just square of x.Compared to 9.")
```

- Is this the end? Can it be improved even further?

Monads - what a strange word.

- What if they can not be avoided?
  - For example input - output operations?

```haskell
inputInt :: Int

inputDiff = inputInt - inputInt

funny :: Int-> Int
funny n = inputInt + n
```

- Haskell uses *programming with actions* to solve this issue.
- Theoretically, we use *thinks* like Functor, Applicative or Monad that comes from the category theory.
  - For *orthodox programmer*, it is just some theory gibberish, for the others, it may provide interesting insight to the problem.
- Informally, a sort of pure functional envelop for non-pure actions.
- Practically, its a set of design patterns solving plenty of situations that are frequently occurring in practice.

IO Monad (1)

- This part is for programmers, that do not care about a theory.
- There is a special type `()` with only value `()` called *unit* type - representing a sort of dummy value.
- All input and output *actions* can be recognized by having `IO` in their type definition.
  - Input: `getLine :: IO String`
  - Output: `putStr :: String -> IO ()`
  - Usually, when we are talking about monads, we say, that they represents some sort of *containers* $\rightarrow$ better intuition for `IO` is: `bake :: Recipe Cake`.
- You can *glue* these actions by syntax construct: `do`.
- How to get value from/to `IO`?
  - There is a syntactic construct in `do` (called bind): `x <- action`, where if `action :: IO a`, then the type of variable x is a.
  - There is a function `return :: a -> IO a`, it can be used to *put* a common value into `IO`.
- Finally, the function `main` has a type: `main :: IO a`
- And that is all, Is it clear?

IO Monad (2)

- Simple example:

```haskell
main = do
    putStrLn "Hello, what's your name?"
    name <- getLine
    let bigName = map toUpper name
    putStrLn ("Hey " ++ bigName ++ ", you rock!")
```

- Now, we can compile it and execute.

```
PS C:\> ghc .\test.hs
[1 of 1] Compiling Main             ( test.hs, test.o )
Linking test.exe ...
PS C:\> .\test.exe
Hello, what's your name?
Marek
Hey MAREK, you rock!
```

IO Monad (3)

- The construct do is just an expression, we can use it in the same way...

```
main = do
    line <- getLine
    if null line
        then return ()
        else do
            print $ reverseWords line
            main
reverseWords :: String -> String
reverseWords = unwords . map reverse . words
```

- You should notice, that return does not end the function like in *common* languages.

```
main = do
    a <- return "hell"
    b <- return "yeah!"
    putStrLn $ a ++ " " ++ b
```

IO Monad (4)

- Interesting question, can we use high order functions with monad `IO`?

```haskell
mySequence :: [IO a] -> IO [a]
mySequence [] = return []
mySequence (ma:mas) = do
  a  <- ma
  as <- sequence mas
  return (a:as)
```

```
ghci> mySequence [getLine, getLine, getLine]
a
b
c
["a","b","c"]
```

- Plenty of functions in: `Control.Monad`.

IO Monad (5)

- For example forM (this is as close to for cycle as you can get in Haskell :-)

```haskell
import Control.Monad

main = do
    lines <- forM [1,2,3] (\a -> do
        putStrLn $ "Give me " ++ show a ++ " line."
        getLine)
    print lines
```

```
Give me 1 line.
Hello
Give me 2 line.
Haskell
Give me 3 line.
programmers
["Hello","Haskell","programmers"]
```

IO Monad (6)

- We can use `IO` monad for working with files.
- For example we can start with: `openFile :: FilePath -> IOMode -> IO Handle`

```haskell
import System.IO

main = do
    handle <- openFile "test.hs" ReadMode
    contents <- hGetContents handle
    putStr contents
    hClose handle
```

- But, there are plenty of other functions:
  - `withFile :: FilePath -> IOMode -> (Handle -> IO a) -> IO a`
  - `readFile :: FilePath -> IO String`
  - `writeFile :: FilePath -> String -> IO ()`, also `appendFile`

IO Monad (7)

- Haskell also have *exceptions* → high order function:

```
catch :: Exception e => IO a -> (e -> IO a) -> IO a
```

```
import System.IO
import System.IO.Error

main = toTry `catch` handler

toTry :: IO ()
toTry = do contents <- readFile "test.txt"
           putStrLn $ "Lines: " ++ show (length (lines contents))

handler :: IOError -> IO ()
handler e = putStrLn "Whoops, had some trouble!"
```

- Functions like `isDoesNotExistError` or `isFullError` to distinguish between *exception*.

IO Monad (8)

- `System.Environment` - for example, for handling command line arguments: `getArgs :: IO [String]`.
- `System.Random` - also random numbers are part of monad `IO`, but here it is complicated.
  - We need to *install* package random: `stack ghci --package random`
  - Now we have:
    `random :: (RandomGen g, Random a) => g -> (a, g)`
    `randomR :: (RandomGen g, Random a) :: (a, a) -> g -> (a, g)`

```
ghci> random (mkStdGen 1) :: (Bool, StdGen)
(True,StdGen {unStdGen = SMGen 4999253871718377453 10451216379200822465})
ghci> randomR (1,6) (mkStdGen 1)
(6,StdGen {unStdGen = SMGen 4999253871718377453 10451216379200822465})
```

- `IO` have also one random generator (`getStdGen`) *stored inside* → we can use functions: `randomIO`, `randomRIO`.
- Be warned. All REAL programmers should stop reading NOW. We will continue with the theory behind monads so we can *outgrowth* the IO monad

Category Theory - why to study? We are programmers!

- Programming is based on math.
  - What kind of math? → geometry, algebra, topology, set theory, type theory, ...
- There are different *kinds* of mathematics → even if developed independently, they share some ideas (for example Curry–Howard correspondence) → **Category theory reveals how different kinds of structures are related to one another.**
  - Category theory is a toolset for describing the general abstract structures in mathematics.
- Category theory is very well suited for programmers → things we normally do overlap with problems category theory is studying.
  - It deals with structure, omitting particulars (abstraction).
  - In its roots, it study composition → holy grail of programming (bigger blocks are composed from components) → composition is crucial in many programming paradigms.
- Haskell have been tapping category theory for a long time, but the ideas can be used also in other languages.

Category Theory - Basics (1)

- *Category C* is algebraic structure consisting of:
  - collection of *objects* - $obj(C)$
  - *arrows (or morphisms or maps)* - $hom(C)$
    - $f : a \to b$ - f is a morphism from a to b.
    - $hom(a, b)$ - hom-set - denotes a set of morphisms from a to b.
- Also, we have a binary operation $\circ$ called *composition* of morphisms.
  - For any three objects a, b and c: $\circ : hom(b, c) \times hom(a, b) \to hom(a, c)$.
  - For any pair of $f : a \to b$ and $g : b \to c$ there exists a composition (composite morphism) written as $g \circ f : a \to c$.
- *Identity*: For every object x, there exists a morphism $1_x : x \to x$ called the identity morphism for x, such that for every morphism $f : a \to b$, we have $1_b \circ f = f = f \circ 1_a$.
- *Associativity*: If $f : a \to b$, $g : b \to c$ and $h : c \to d$ then $h \circ (g \circ f) = (h \circ g) \circ f$.

Category Theory - Basics (2)

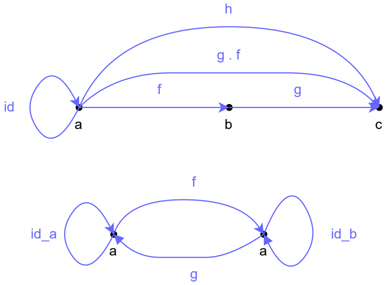


Figure: Are these schematics for categories?

- Usually, a schematics for a category is a *multigraph* where objects are verticies, and morphisms are oriented edges.
- When we are talking about categories in programming, most common example are types and functions.
  - For example our figure depicts functions:

    ```
    f :: a -> b
    g :: b -> c
    ```
  - Function $g \circ f$ can be defined as:

    ```
    gf :: a -> c
    gf = \x -> g (f x)
    ```

Category Theory - Monoid

- Important algebraic structure in category theory is *monoid*.
  - Part of set theory, used even before the whole category theory thing $\rightarrow$ we already know this term from set theory $\rightarrow$ what is their relation?
- A set S with $* : S \times S \rightarrow S$ (multiplication) is a **monoid** M if it satisfies:
  - *Associativity*: For all a, b and c in S, the equation $(a * b) * c = a * (b * c)$ holds.
  - *Unit element*: There exists an element e (unit) in S such that for every element a in S, the equalities $e * a = a$ and $a * e = a$ hold.
- An individual monoid $(M, *, e)$ *can be* a category C where:
  - the collection of objects $obj(C)$ is single object - M;
  - the collection of morphisms $hom(C)$ is set M itself, which mean, each element in set M is a morphism in category C;
  - the composition operation $\circ$ of C is $*$ - since each morphism in C is element in M, the composition of morphisms is just the multiplication of elements;
  - the identity morphism of C is unit element e
- In this way, since M,$*$ and e satisfies the monoid laws, apparently the category laws are satisfied.

Monoid in Haskell (1)

- Stop this theory gibberish, what it has to do with mentioned practical examples?
- Let's define *monoid* in Haskell.

```
-- Data.Monoid
-- class Semigroup a => Monoid a where
class Monoid m where
    mempty :: m
    mappend :: m -> m -> m
    mconcat :: [m] -> m
    mconcat = foldr mappend mempty
```

- What about the rules for monoid?

```
mempty `mappend` x = x
x `mappend` mempty = x
(x `mappend` y) `mappend` z
    = x `mappend` (y `mappend` z)
```

- (This is embarrassing.) Haskell can not enforce them, programmer is kindly asked to obey them.

Monoid in Haskell (2)

- Plenty of types are instances of `Monoid`.

```
instance Monoid [a] where
    mempty = []
    mappend = (++)
```

  - `All`, `Any`, `First`, `Last`, `Maybe`, `Ordering`, `IO`, `Sum`, `Product`, ...

- Do you remember our logging example? We can modify our logging function like this:

```
applyLog :: (Monoid m) => (a,m) -> (a -> (b,m)) -> (b,m)
applyLog (x,log) f = let (y,newLog) = f x in (y,log `mappend` newLog)
```

```
*Main> (2,"Initial value 2.") `applyLog` compute `applyLog` isItEnough
(False,"Initial value 2.Just square of x.Compared to 9.")
```

- The result is the same, but **now**... (wait for it:-)

Monoid in Haskell (3)

- We can use the same function `applyLog` with all types that are instances of `Monoid`.
- Let's say, we want to log just some events.

```
addMore :: Int -> (Int, Maybe String)
addMore x
    | x == 2 =  (x+1, Just "Nice.")
    | x == 1 =  (x+1, Just "More!")
    | otherwise = (x+1, Nothing)
```

```
*Main> (1,Nothing) `applyLog` addMore  `applyLog` addMore `applyLog` addMore
(4,Just "More!Nice.")
```

- Is it better then before? We all agree that it is. Right ?!?

Functor (1)

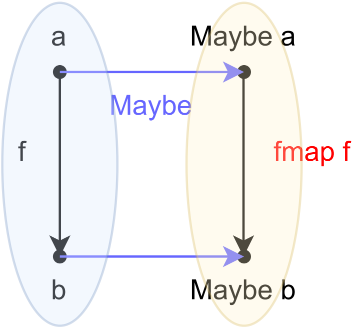
- Motivation
  - Our original goal was to find some nice (design) patterns for frequently occurring problems.
  - Lets say, we found one, what next? $\rightarrow$ prepare abstract solution capturing the idea $\rightarrow$ apply it to solve other problems.
  - In terms of categories, this abstraction is captured by a category and we need to *transfer* it to other categories.
- **Functor** is a mapping between categories that preserve a structure $\rightarrow$ it preserve identity morphisms and composition of morphisms.
- Let C and D be categories. A functor F from C to D is a mapping (function) that:
  - associates each x in $obj(C)$ to an object $F(X)$ in $obj(D)$,
  - associates each morphism $f : X \rightarrow Y$ in C to a morphism $F(f) : F(X) \rightarrow F(Y)$ in D such that the following two conditions hold:
    - $F(id_x) = id_{F(x)}$ for every x in $obj(C)$,
    - $F(g \circ f) = F(g) \circ F(f)$ for all morphisms $f : X \rightarrow Y$ and $g : Y \rightarrow Z$ in C

Functor (2)

But how to implement functors in Haskell?

- Lets say, we *just want to add* `Maybe` for capturing errors → We want to map our structure to *Maybe category* → We need a **functor**.
- First, we need to map *objects* (types) → type constructor `Maybe`
- Second, we need to map *morphisms* (functions):

```
fmap :: (a->b) -> (Maybe a -> Maybe b)
fmap f (Just x) = Just (f x)
fmap _ Nothing = Nothing
```



Functor (3)

- Let's try to generalize this approach. We introduce new type class: `Functor`

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

- What is f in the definition? → A type constructor with kind `* -> *` and a method `fmap`.

- Kind in haskel is a type of the type.

```
 *Main> :kind Int
Int :: *
*Main> :kind Maybe
Maybe :: * -> *
```

- Note, the result is similar to operator $, we can even use it in the same way.
- There is even an operator: `<$> = fmap`

Functor (4)

- So, we can do:

```
*Main> (+1) $ (*2) $ (+3) $ 1 --looks good, but it's cheating...
9
*Main> (+1) `fmap` ((*2) `fmap` ((+3) `fmap` (Just 1)))
Just 9
*Main> (+1) <$> (*2) <$> (+3) <$> (Just 1) -- fmap on ->
Just 9
```

- What about operations like +?

```
*Main> (+) <$> (Just 1) -- Maybe (Int -> Int)
```

- Again, plenty of types are instances of Functor.
  - List `[]` here: `fmap = map`
  - `->`, First, Last, Sum, Product, Min, Max, Identity, IO, ST a, Array i,...

Functor (5)

- What about `Either a b`, can it be a functor?

```haskell
-- * -> * -> *
data Either a b = Left a
                | Right b
```

- Not really, but `Either a` is OK!

```haskell
instance Functor (Either a) where
    fmap _ (Left x) = Left x
    fmap f (Right y) = Right (f y)
```

- When we are defining a function, we are using `->`. What is it? Can it be a functor? How to define $fmap$ then?

Functor (6)

- Is that all? What about the rules from the functor definition?

```
fmap id == id -- Identity
fmap (f . g) == fmap f . fmap g -- Composition
```

- Again, programmer is kindly asked to obey them.
- It does not obey mentioned rules, but it will *work*.

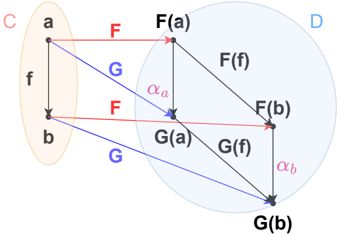
```
data CMaybe a = CNothing | CJust Int a

instance Functor CMaybe where
    fmap f CNothing = CNothing
    fmap f (CJust counter x) = CJust (counter+1) (f x)
```

- **Endofunctor** is a functor where the source and the target category is the same.
  - Strictly speaking, the `Functor` class represents endofunctors on the category of Haskell types and functions.
  - Endofunctors are interesting because they do a good job of representing structures inside categories that work for *any* object.

Fun with functors I

- Are we done with functors? (NO! The fun barely started:-)
- Category of categories - Cat
  - Functors can be composed: if we have $F : C \to D$ and $G : D \to E$ it is easy to define new functor $H : C \to E$ as $G \circ F$
  - We can always define an identity functor.
- **Natural transformation** defines a relation between functors. For $F : C \to D$ and $G : C \to D$, the natural transformation $\alpha : F \Rightarrow G$ is a family of morphisms (from D) where:
  - $\forall X \in obj(C)$, we pick a morphism $\alpha_X : F(X) \to G(X)$ in D (called the component of α_X at X.
  - $\forall f : X \to Y \in hom(C)$, $\alpha_Y \circ F(f) = G(f) \circ \alpha_X$ (naturality square or condition).

Fun with functors II



- Logical next step is: **Category of functors** $[C, D]$ or D^C
  - Objects $obj(D^C)$ are functors from C to D
  - Morphisms $hom(D^C)$ are natural transformations between those functors.
  - Identity $id_F : F \Rightarrow F$ - maps each functor to itself.
  - Composition of $\alpha : F \Rightarrow G$ and $\beta : G \Rightarrow H$ is $(\beta \circ \alpha) : F \Rightarrow H$, defined as *composition* of morphisms in D: $(\beta \circ \alpha)_X : F(X) \Rightarrow H(X) = (\beta_X : G(X) \rightarrow H(X)) \circ (\alpha_X : F(X) \rightarrow G(X))$ (so they obey the associativity).

Fun with functors III

- If we use the same category, we get: **Category of endofunctors** C^C
  - Haskell functors are in fact endofunctors on category of types and functions. What will be a natural transformation? $\rightarrow$ Polymorphic function with type:

    ```
    alpha :: F a -> G a -- for all a
    ```
  - Note: Most such polymorphic functions are natural transformations.
  - Note: We can not really change *the value*, just its *computational context*.
- Example

```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:xs) = Just x
```

```
*Main> (safeHead . fmap (+1)) [1]
Just 2
*Main> (fmap (+1) . safeHead) [1]
Just 2
```

  - What about the *naturality square* $\rightarrow$ It is always satisfied! (Nice:-).

    ```
    (alpha . fmap f) = (fmap f . alpha)
    ```

Monoidal Categories (1)

- A **product category** $C \times D$ is a category where:
  - objects are pairs: (A, B) where $A \in obj(C), B \in obj(D)$
  - there is a morphisms $(f, g) : (A_1, B_1) \to (A_2, B_2)$ for all pairs of morphisms: $f : A_1 \to A_2$ from C and $g : B_1 \to B_2$ from D
  - composition: $(f_2, g_2) \circ (f_1, f_2) = (f_2 \circ f_1, g_2 \circ g_1)$
  - identity: $1_{(A,B)} = (1_A, 1_B)$
- A **bifunctor** is the mapping from a product category $C \times D$ to category E, denoted: $F : C \times D \to E$.
  - Again in haskell it is implemented as $p : C \times C \to C$.

    ```
    class Bifunctor p where
        bimap :: (a -> b) -> (c -> d) -> p a c -> p b d
        first :: (a -> b) -> p a c -> p b c
        second :: (b -> c) -> p a b -> p a c
    ```
  - Good example is: `Bifunctor Either` or `Bifunctor (,)`.

Monoidal Categories (2)

- A **monoidal category** $(C, \otimes, I)$ is a category C equipped with:
  - a bifunctor $\otimes : C \times C \to C$ called monoidal product or tensor product;
  - an object I called (monoid, tensor) unit or identity object;
  - moreover, it needs to be equipped with natural transformations to satisfy monoid laws:
    - associator: $\alpha_{X,Y,Z} : (X \otimes Y) \otimes Z \Rightarrow X \otimes (Y \otimes Z)$, where $X, Y, Z \in obj(C)$
    - left unitor: $\lambda_A : I \otimes A \Rightarrow A$ and right unitor: $\rho_A : A \otimes I \Rightarrow A$
- Category theory gives us a new way, how to define a monoid. If we have a monoidal category $(C, \otimes, I)$ then any $M \in obj(C)$ with two morphisms:
  - $\mu : M \otimes M \to M$ (multiplication)
  - $\eta : I \to M$ (unit)

  is a monoid.
- Hold that thought, we will use right after *applicative*...

Applicative (1)

- On our path to monads, we can continue with different types of *monoidal functors*, but as programmers we have something more intuitive: *Applicative functor*.
  - Informally, monoidal functors are functors between two monoidal categories that preserves *monoidal structure*.
  - Applicative functors are the programming equivalent of lax monoidal functors with tensorial strength (if it means something:-).
  - Applicative functors allow for functorial computations to be sequenced (unlike plain functors), but don't allow using results from prior computations in the definition of subsequent ones (unlike monads).
- **Applicative functor** is a functor with the ability to apply functor-wrapped functions with functor-wrapped values. It is a functor with two

```haskell
class Functor f => Applicative f where
  pure :: a -> f a
-- $    ::    (a -> b) ->   a ->   b
-- fmap ::    (a -> b) -> f a -> f b
  (<*>) :: f (a -> b) -> f a -> f b
```

Applicative (2)

- Again, it must preserve some additional rules.
  - *Identity*: `pure id <*> v = v`
  - *Composition*: `pure (.) <*> u <*> v <*> w = u <*> (v <*> w)`
  - *Homomorphism*: `pure f <*> pure x = pure (f x)`
  - *Interchange*: `u <*> pure y = pure ($ y) <*> u`
- We can notice, that if we have a type from `Applicative`, we have also `Functor`

```
fmap f x = (pure f) <*> x
```

```
instance Applicative Maybe where
    pure x = Just x
    (Just f) <*> (Just x) = Just (f x)
    _ <*> _ = Nothing
```

```
*Main> (Just (+)) <*> (Just 1) <*> (Just 2)
Just 3
*Main> (+) <$> (Just 1) <*> (Just 2)
Just 3
```

Monads - Category Way (1)

- We defined a monoidal category $\rightarrow$ but endofunctor in a endofunctor category can be monoidal too.
  - Such *Monoid in the category of endofunctors is a* ***monad***.
- Formally, for category C, a monad F is an endofunctor $F : C \rightarrow C$ equipped with two natural transformations:
  - monoid multiplication $\odot$ or μ: $\odot : F(F) \Rightarrow F$ (for clarity denotated: $F \odot F \Rightarrow F$) - for each $X \in obj(C)$, $\odot$ maps $F(F(X) \rightarrow F(X)$;
  - monoid unit η, $\eta : 1_C \Rightarrow F$, 1_C is in fact identity functor, $\forall X \in C : 1_C(X) = X$, so η is in fact mapping: $X \rightarrow F(X)$.
  - Moreover, it preserve following rules:
    - Associativity preservation $\alpha : (F \odot F) \odot F \equiv F \odot (F \odot F)$
    - Left unit preservation $\lambda : \eta \odot F \equiv F$
    - Right unit preservation $\rho : F \equiv F \odot \eta$
- So, now is the moment when the theory should compose together and shine:-)

Monads - Category Way (2)

1. Haskell type class `Functor` represents in fact *endofunctors* on category of Haskell types and functions (H). We can define a *category of endofunctors* H^H.
2. In this category, objects are *instances* of `Functor` (for example F and G) and morphisms are natural transformations between then $\rightarrow$ they are polymorphic functions:
   `alpha :: F a -> G a`
3. If we want to make our category H^H a monoidal category, we need to introduce a tensor product $(H^H \times H^H) \rightarrow H^H$ and tensor unit (object from H^H). One natural way to do that, is to define:
   - tensor product as **composition** of endofunctors: $F \circ G$ (it is associative);
   - tensor unit as identity endofunctor: Id.
4. To define a monoid based H^H on we need to pick an object - endofunctor T along with two morphisms (natural transformations in H):
   - $\mu : T \otimes T \rightarrow T$ - function: `join :: T (T a) -> T a`
   - $\eta : I \rightarrow M$ (unit) - function `return :: a -> T a`
5. Finally, such endofunctor T is a **monad**! $\rightarrow$ It is a monoid in the category of endofunctors.

Monads - Programmers Way (1)

- New functions are produced like a **composition** of functions → important abstraction mechanism. `(.) :: (b -> c) -> (a -> b) -> a -> c`
- The ordering of functions does not matter, we can introduce:
  `(>.>) :: (a -> b) -> (b -> c) -> a -> c`
- We want to have something similar to that for our `Functor` class. How the functions from our examples looked liked?

```
eval :: Expr -> Maybe Int
compare :: Int -> Maybe Bool
```

- So, to be able to *compose* such functions, we need something like:
  `(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c`
- Consider, we have an operator $>>=$ (bind): `(>>=) :: m a -> (a -> m b) -> m b`
- Then it is easy, operator $>=>$ (Fish operator, Klesli category) can be defined as:

```
f (>=>) g = \ a -> let mb = f a
                   in mb >>= g
```

Monads - Programmers Way (2)

- OK, we have eliminated some unnecessary staff, but we still need: `(>>=) :: m a -> (a -> m b) -> m b`, right?
- That is precisely how monads are defined in Haskell.

```
class Applicative f => Monad f where
  (>>=) :: f a -> (a -> f b) -> f b
  return :: a -> f a
```

- Again, if we have `Monad`, we also have `Functor` and `Applicative`. The prove, is not that obvious as before.

```
fmap fab ma   =  ma >>= (\x -> return (fab x)) -- (return.fab)
pure a        =  return a
mfab <*> ma   =  mfab >>= (\ fab -> ma >>= (return . fab))
```

Monads - Programmers Way (3)

- Alternatively, if we want to define >>= and we know that f is a `Functor`. Bind operator can be defined:

```
(>>=) :: f a -> (a -> f b) -> f b
ma >>= f = join (fmap f ma)
-- in API: join :: Monad m => m (m a) -> m a
join :: m (m a) -> m a
```

- So, in theory a monad can be also defined by functions: $join$ and $return$ $\rightarrow$ Wait, that's our μ and η morphisms in monad definition. $\rightarrow$ That's precisely where we ended up following the category theory!
- We can easily define $=<<$ that just swaps the parameters of bind:

```
-- $    ::                (a -> b) -> a -> b
-- fmap ::                (a -> b) -> f a -> f b
--(<*>) ::              f (a -> b) -> f a -> f b
(=<<)   :: Monad m => (a -> m b) -> m a -> m b
f =<< x = x >>= f
```

Programming with actions (1)

- Now, we can chain actions better.

```
*Main> (Just 1) >>= (\x-> return (x+1))
Just 2
*Main> (Just (+)) >>= (\y -> Just (y 1 2)) >>= (\x -> return (x+1))
Just 4
*Main> Just 3 >>= (\x -> Just "!" >>= (\y -> Just (show x ++ y)))
Just "3!"
*Main> Just 3 >>= \x -> Just "!" >>= \y -> Just (show x ++ y)
Just "3!"
```

- We can even solve our original problem!

Programming with actions (2)

- Solving *maybe* expressions with *monads*.

```haskell
eval :: Expr -> Maybe Int
eval (Num x) = return x
eval (Div x y) = eval x >>= (\x' -> eval y >>= (\y' -> betterDiv x' y'))
eval (Add x y) = eval x >>= \x' -> eval y >>= \y' -> return ( x'+ y')
eval (Mul x y) = eval x >>=
                 \x' -> eval y >>=
                 \y' -> return ( x'* y')
eval (Sub x y) = do x' <- eval x
                    y' <- eval y
                    return ( x'- y')
```

List Monad (1)

- Nice *example* of a monad is the list. Informally, required operations are implemented:

```
myFmap :: (a -> b)   -> [a] -> [b]
myFmap = map

myApply :: [a -> b]   -> [a] -> [b]
myApply fs xs = [f x | f <- fs, x <- xs]

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

- Now, we can observe, what we *can do* with such defined operators.

```
*Main> (+1) <$> [1,2,3]
[2,3,4]
*Main> (+) <$> [1,2,3] <*> [1,2,3]
[2,3,4,3,4,5,4,5,6]
*Main> [1,2] >>= \n -> ['a','b']
             >>= \ch -> [(n,ch)]
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
*Main> [3,4,5] >>= (return . (+1))
               >>= (return . (*2))
[8,10,12]
```

IO Monad - just to remind you

- In previous part, we have introduced a mechanism how *actions* can be chained → nicer way how to write it.
- But we have started with the idea, that impure actions (manipulating with *state*) will be solved with monads.
- We already know `IO Monad` that solves input - output operations.

```haskell
-- inputLine :: String
getLine :: IO String
putStr :: String -> IO ()

do x <- getLine
   putStr x -- y <- putStr x, y == ()

ready :: IO Bool
ready = do c <- getChar
           return (c == 'y')
```

State Monad (1)

- How does it work? The idea is captured in more general monad that captures state.
- Lets first focuse on the idea → state manipulation can be captured like a function taking original state and producing a pair (some value, new state).

```
type SimpleState s a = s -> (s, a)

retSt :: a -> SimpleState s a
--retSt a s = (s,a)
retSt a = \s -> (s,a)
```

- Now, lets create a simple *input* containing a list of integers (our state is just this list).

```
type ListInput a = SimpleState [Int] a

readInt :: ListInput Int
readInt stateList = (tail stateList, head stateList)
```

State Monad (2)

- Finally, lets try to make a function chaining actions (like >>=).

```
bind :: (s -> (s,a))             -- SimleState s a
     -> (a -> (s -> (s, b))) -- a -> SimpleState s b
     -> (s -> (s, b))         -- SimpleState s b
bind step makeStep oldState = -- Why 3 parameters?
    let (newState, result) = step oldState
    in  (makeStep result) newState
```

- Finally, we can bind actions as with monads.

```
*Main> (readInt `bind` \a->readInt `bind` (\b->retSt (a+b))) [1,2,3]
([3],3)
```

- In our example, we have created a function defining what to do with the input. When it is executed it *bakes* the result. If provided the same *ingredients*, it *bakes* the same result.

State Monad (3)

- What if we want to realy make it a part of `Monad` type class (it will not work for type synonym)?

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

readInt' :: State [Int] Int
readInt' = State {runState = \s->(tail s, head s)}

instance Functor (State s) where
    fmap f m = State $ \s-> let (s',a) = runState m s in (s',f a)

instance Applicative (State s) where
    pure a = State (\s->(s,a))
    f <*> m = State $ \s-> let  (s',f') = runState f s
                                (s'',a) = runState m s' in (s'',f' a)
instance Monad (State s ) where
    return a = State (\s->(s,a))
    m >>= k = State $ \s -> let (s',a) =  runState m s  in runState (k a) s'
```

State Monad (4)

- We can even use do syntax now.

```
add :: State [Int] Int
add = do x<-readInt'
         y<-readInt'
         return (x+y)
```

- Examples, how to use this state monad:

```
*Main> runState (readInt' >>= \a->readInt' >>= (\b->return (a+b))) [1,2,3]
([3],3)
*Main> runState add [1,2,3]
([3],3)
```

- Finally, assuming we have RealWorld, we ca define type IO as:

```
type IO a = State RealWorld a
--getChar :: RealWorld -> (RealWorld, Char)
--main :: RealWorld -> (RealWorld, ())
```

Stacking Monads (1)

- What if we want to use *several* monads → We want to use *state* and `Maybe` → monad transformers (`Control.Monad.Trans`).
- For example, we will use wrapper:

```
newtype MaybeT m a = MaybeT { runMaybeT :: m (Maybe a) }
instance Monad m => Monad (MaybeT m) where
  return = MaybeT . return . Just
  -- (>>=) :: MaybeT m a -> (a -> MaybeT m b) -> MaybeT m b
  x >>= f = MaybeT $ do
    maybe_value <- runMaybeT x
    case maybe_value of
      Nothing    -> return Nothing
      Just value -> runMaybeT $ f value
```

Stacking Monads (2)

- For practical purposes, we need *lift* function - it promotes base monad computations to combined monad.
  - It is similar to `liftM :: Monad m => (a -> b) -> (m a -> m b)` method for combined monad.
- For example, we will use wrapper:

```
class MonadTrans t where
  lift :: (Monad m) => m a -> t m a

instance MonadTrans MaybeT where
  lift = MaybeT . (liftM Just)
```

Stacking Monads (3)

- Example:

```haskell
import Control.Monad.Trans.Maybe
import Control.Monad.IO.Class (liftIO)
import Text.Read

data Person = Person {name::String, age::Int} deriving Show

askPersonT :: MaybeT IO Person
askPersonT = do
  name   <- liftIO $ putStr "Name? " >> getLine
  age    <- MaybeT $ fmap readMaybe $ putStr "Age? " >> getLine
  return $ Person name age

doIt = do result <-runMaybeT askPersonT
          print result
```

Arrays in Haskell

- Like in other languages Haskell has arrays.
- Arrays (where we can get i^{th} element in $O(1)$) are best choice for some algorithms.
- Boxed (non-strict) arrays support lazy evaluation.
- Unboxed (strict) - just values, only basic types, closer to $memory\ block$.
- Arrays are in package $array$.

	Immutable	IO monad	ST monad
	`instance IArray a e`	`instance MArray a e IO`	`instance MArray a e ST`
Boxed	Array DiffArray	IOArray	STArray
Unboxed	UArray DiffUArray	IOUArray StorableArray	STUArray

Table: Comparison of an different $arrays$ in Haskell

Immutable Array (1)

- Immutable arrays are in modules: `Data.Array` or `Data.Array.IArray`
- All these *arrays* use the same indexing.

```
class  (Ord a) => Ix a  where
  range        :: (a,a) -> [a]
  index        :: (a,a) a -> Int
  inRange      :: (a,a) -> a -> Bool
```

- Then (based on imported array *type*), we create an array:

```
array :: (Ix a) => (a,a) -> [(a,b)] -> Array a b
listArray :: Ix i => (i, i) -> [e] -> Array i e
```

```
squares = array (1,100) [(i, i*i) | i <- [1..100]]
listToArray = listArray (0,5) [8,4,9,6,7,1]
```

Immutable Array (2)

- Accessing arrays (works also for `IArray`):

```
(!) :: (Array a e, Ix i) => a i e -> i -> e
bounds :: (Array a e, Ix i) => a i e -> (i, i)
indices :: (Array a e, Ix i) => a i e -> [i]
elems :: (Array a e, Ix i) => a i e -> [e]
```

- Incremental array updates (works also for `IArray`):

```
(//) :: (Array a e, Ix i) => a i e -> [(i, e)] -> a i e
```

```
ghci> listArray (0,5) [8,4,9,6,7,1] // [(1,0),(2,0)]
array (0,5) [(0,8),(1,0),(2,0),(3,6),(4,7),(5,1)]
```

- Derived arrays (amap requires `IArray`):

```
amap :: (IArray a e', IArray a e, Ix i) => (e'->e) -> a i e' -> a i e
ixmap :: (Array a e, Ix i, Ix j) => (i, i) -> (i->j) -> a j e -> a i e
```

Mutable Array (1)

- Class of mutable array types:

```
class Monad m => MArray a e m ... --array: (a i e), index: Ix i
```

- We need a monad to preserve a state: `ST s` or `IO`.
- Constructing mutable arrays:

```
newArray :: (MArray a e m, Ix i) => (i, i) -> e -> m (a i e)
newListArray :: (MArray a e m, Ix i) => (i, i) -> [e] -> m (a i e)
```

- Reading and writing mutable arrays:

```
readArray :: (MArray a e m, Ix i) => a i e -> i -> m e
writeArray :: (MArray a e m, Ix i) => a i e -> i -> e -> m ()
```

- Derived arrays

```
mapArray::(MArray a e' m,MArray a e m, Ix i)=>(e'->e)-> a i e'->m (a i e)
mapIndices::(MArray a e m,Ix i,Ix j)=>(i, i)->(i->j)->a j e->m (a i e)
```

Mutable Array (2)

- Deconstructing mutable arrays:

```
getBounds :: (MArray a e m, Ix i) => a i e -> m (i, i)
getElems :: (MArray a e m, Ix i) => a i e -> m [e]
getAssocs :: (MArray a e m, Ix i) => a i e -> m [(i, e)]
```

- Conversions between mutable and immutable arrays:

```
freeze :: (Ix i, MArray a e m, IArray b e) => a i e -> m (b i e)
thaw :: (Ix i, IArray a e, MArray b e m) => a i e -> m (b i e)
```

- Let's use monad `ST` to preserve the state.
- Now, we have: `data STArray s i e`, it will be an instance of `MArray (STArray s) e (ST s)`
- Safe way to create and work with mutable array:

```
runSTArray :: (forall s. ST s (STArray s i e)) -> Array i e
```

It will return immutable array at the end (it will `thaw` the original array).

Mutable Array (3)

- Example how to use mutable array:

```
modify :: Array Int Int -> Array Int Int
modify inputArray = runSTArray $ do
    let end = (snd . bounds) inputArray
    stArray <- thaw inputArray
    forM_ [1 .. end] $ \i -> do
        val <- readArray stArray i
        when (val<0) $ do
            writeArray stArray i 0
    return stArray
```

```
ghci> modify $ listArray (0,3) [8,-4,-9,1]
array (0,3) [(0,8),(1,0),(2,0),(3,1)]
```

Conclusion

- In Haskell, monads are a *sort of* functional envelop for *in-pure* functions.
- Functions like *bind*, *join* or *fmap* allows us to work with these monads.
    - On the first sight, we can recognize a function working with input/output → it will have `IO` in the type definition.
    - We can use the *same* design patterns for *all* monads.
- Strictly speaking, we can forget all about the theory and just use `do` if it is *a monad*.

Thank you for your attention

Marek Běhálek

VSB – Technical University of Ostrava

marek.behalek@vsb.cz

November 4, 2022