IA014: Advanced Functional Programming

6. Monads

Jan Obdržálek obdrzalek@fi.muni.cz

Faculty of Informatics, Masaryk University, Brno

What a Monad Is Not

The following claims are all **false**!

- Monads are impure.
- Monads are about effects.
- Monads are about state.
- Monads are about sequencing.
- Monads are about IO.
- Monads are dependent on laziness.
- Monads are a "back-door" in the language to perform side-effects.
- Monads are an embedded imperative language inside Haskell.
- Monads require knowing abstract mathematics.

See `http://www.haskell.org/haskellwiki/What_a_Monad_is_not`

Where's the catch

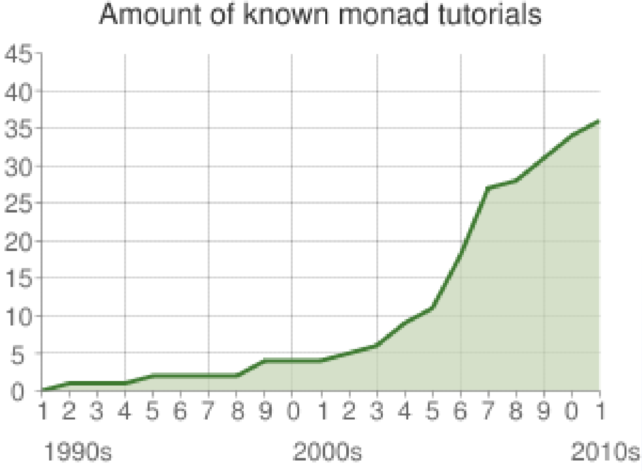
The "Problem" with Monads

1. monads are *abstract*
2. monads are used in *several different capacities*
3. analogies make things worse
   they are created by people, who *already understand* monads

Benefits of Monads

1. modularity
2. flexibility
3. isolation

History of monad tutorials



https://www.haskell.org/haskellwiki/Monad_tutorials_timeline

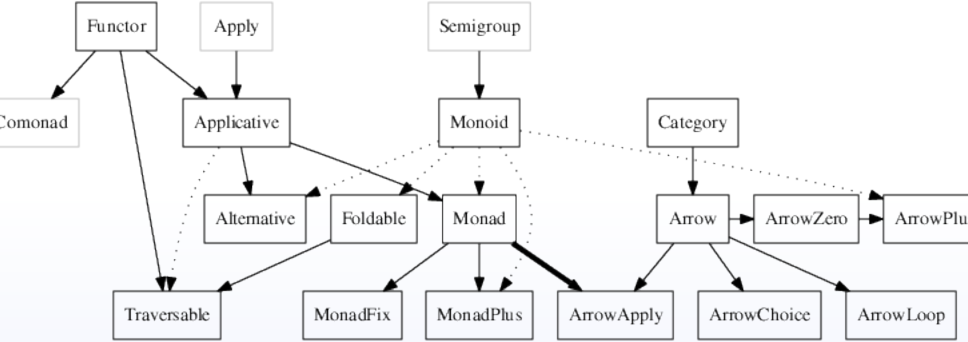
Eightfold Path to Monad Satori

(by Stephen Diehl)

1. Don’t read the monad tutorials.
2. No really, don’t read the monad tutorials.
3. Learn about Haskell types.
4. Learn what a type class is.
5. Read the Typeclassopedia.
6. Read the monad definitions.
7. Use monads in real code.
8. Don’t write monad-analogy tutorials.

`http://dev.stephendiehl.com/hask/#monads`

Haskell class diagram



https://www.haskell.org/haskellwiki/Typeclassopedia

More fun with Functors

Functors

As defined in Prelude:

```
class  Functor f  where
    fmap        :: (a -> b) -> f a -> f b
    (<$)        :: a -> f b -> f a
    (<$)        =  fmap . const
```

- instances: types which can be `map`ped over
- the `<$` operator replaces all locations in the input by the same value
- `<$` default implementation can also be written as:

```
v <$ c = fmap (const v) c
```

Functors as containers

`Functor` represents a *container* (of some sort), for which we can apply a function to each element in the container.

examples

```
> fmap (\x -> x + 1) [1,2,3]
[2,3,4]
> fmap (\x -> x + 1) (Just 41)
Just 42
> fmap (\x -> x + 1) Nothing
Nothing
> fmap (\x -> x + 1) (Right 41)
Right 42
> fmap (\x -> x + 1) (Left False)
Left False
```

Note aside: `Either`

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

Functors as computational contexts

```
instance  Functor Maybe  where
    fmap _ Nothing       = Nothing
    fmap f (Just a)      = Just (f a)
```

- the computational context view:
  - context with possible failure
  - `Just a` value – result of a computation
  - `Nothing` – failure

```
instance Functor ((->) e) where
    fmap f g = \x -> f (g x)
```

- comments:
  - `fmap ::  (a -> b) -> (e -> a) -> (e -> b)`
  - `(->) e`should be read as `(e ->)` (the `(1+)` analogy)
- computational context view:
  - context in which the value of type `e` is available in a read-only fashion
- container view:
  - set of values of `a`, indexed by values of `e`

Functor laws

```
fmap id  ==  id
fmap (f . g)  ==  fmap f . fmap g
```

- part of the definition of a mathematical functor (category theory)
- ensure that `fmap g` does not change *structure*, only the *contents* of the container
- ensure that `fmap g` changes a *value*, not its `context`
- laws are not enforced by the compiler
- lists, `Maybe` and `IO` satisfy these laws

Breaking the laws

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

```
> fmap id [1,2,3]
[2,3]
> fmap (+1) $ fmap (+1) [1,2,3]
[5]
> fmap ((+1) . (+1)) [1,2,3]
[4,5]
```

- it is not a good idea to break the laws
- breaks the intuition about meaning/behaviour of various classes

Functor Lifting

```
fmap        :: (a -> b) -> f a -> f b
```

- what if we *partially apply* `fmap` to `g`?
  - it takes `g` of type `a -> b`
  - and returns a function `(fmap g)` of type `f a -> f b`
- we *transform* a “normal” function `g` to a one operating on containers
- this transformation is called *lifting*

Applicative Functors

Functions in containers

Mapping functions of multiple arguments

```
> let a = fmap (+) [1,2,3]
```

We create a list of functions `[(+)1, (+)2, (+)3]`

```
> :t a
a :: [Integer -> Integer]
> fmap (\f -> f 10) a
[11,12,13]
```

More examples:

```
> let b = fmap (+) (Just 42)
> let c = fmap (+) Nothing
> let d = fmap (\x y -> x + y) (Right 42)
```

Applying functions in containers

Remember:

```
> let b = fmap (+) (Just 42)
> fmap (\f -> f 1) b
Just 43
```

What about *applying a function in a container to a value in a container*?

```
> (Just (41+)) (Just 2)
```

FAIL!

Solution:

```
(<*>) :: Maybe (a->b) -> Maybe a -> Maybe b
(Just f) <*> v = fmap f v
Nothing  <*> _ = Nothing
```

Let's try:

```
> Just (41+)) <*> (Just 2)
Just 43
```

The Applicative class

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

What is pure?

- places a value into a context/container!

the laws

```
pure id <*> v = v
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)
pure f <*> pure x = pure (f x)
u <*> pure y = pure ($ y) <*> u

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

Lists 1/2

Two possible views:

1. context view
2. collection view

Context view

- elements represent multiple results of a nondeterministic computation
- default view in HASKELL

```
> pure (+) <*> [2,3,4] <*> pure 4
[6,7,8]
```

The definition:

```
instance Applicative [] where
    pure x = [x]
    gs <*> xs  = [ g x | g <- gs, x <- xs ]
```

Lists 2/2

collection view

- list is an ordered collection of elements
- the application of functions to elements is pointwise
- we have to define a new type

```
newtype ZipList a = ZipList { getZipList :: [a] }

instance Applicative ZipList where
  pure = undefined   -- exercise
  (ZipList gs) <*> (ZipList xs) = ZipList (zipWith ($) gs xs)
```

Applicative comments

Some instances

- `[]`
- `ZipList`
- `Maybe`
- `(Either e)`
- `((->) a)`
- `IO`

Further reading

C. McBride, R. Paterson: *Applicative Programming with Effects*
(Journal of Functional Programming 18:1, 2008)

Monads: the tutorial

Based on the *very first* monad tutorial, by Phil Wadler.

Basic evaluator

```haskell
data Exp = Plus  Exp Exp
         | Minus Exp Exp
         | Times Exp Exp
         | Div   Exp Exp
         | Const Int

eval :: Exp -> Int
eval (Plus  e1 e2) = (eval e1) + (eval e2)
eval (Minus e1 e2) = (eval e1) - (eval e2)
eval (Times e1 e2) = (eval e1) * (eval e2)
eval (Div   e1 e2) = (eval e1) `div` (eval e2)
eval (Const i)     = i

answer = (Div (Div (Const 1972) (Const 2)) (Const 23))
err    = (Div (Const 1) (Const 0))
```

```
> eval answer
42
> eval err
*** Exception: divide by zero
```

Variation 1: adding error handling

```
data M1 a = Raise Exception | Return a deriving Show
type Exception = String

evalE :: Exp -> M1 Int
-- Plus, Minus, Times cases omitted.
evalE (Div e1 e2) =
   case evalE e1 of
      Return a ->
        case evalE e2 of
          Return b -> if b == 0 then Raise "division by 0"
                               else Return (a `div` b)
          Raise s -> Raise s
      Raise s -> Raise s
evalE (Const i)  = Return i
```

```
> eval answer
Ok 42
> eval err
Raise "division by 0"
```

Variation 2: adding state

Task: Count the number of evaluation steps.

```
type M2 a    = State -> (a, State)
type State = Int

evalS :: Exp -> M2 Int
-- Plus, Minus, Times cases omitted.
evalS (Div e1 e2) x = let (a, y) = evalS e1 x in
                      let (b, z) = evalS e2 y in
                      (a `div` b, z+1)
evalS (Const i)   x = (i, x)
```

```
> evalS answer 0
(42,2)
```

Variation 3: adding output

Task: Show the evaluation steps

```
type Output = String

eval0 :: Exp -> M3 Int
-- Plus, Minus, Times cases omitted.
eval0 (Div e1 e2) = let (a, x) = eval0 e1 in
                    let (b, y) = eval0 e2 in
                    (a `div` b,
                        x ++ y ++ line (Div e1 e2) (a `div` b))
eval0 (Const i)   = (i, line (Const i) i)

line :: Exp -> Int -> Output
line e a = show e ++ "=" ++ show a ++ "\n"
```

Variation 3: example run

Test data:

```
answer = (Div (Div (Const 1972) (Const 2)) (Const 23))
err    = (Div (Cost 1) (Const 0))
```

Results:

```
> Eval0 answer
(42,"Const 1972=1972
Const 2=2
Div (Const 1972) (Const 2)=986
Const 23=23
Div (Div (Const 1972) (Const 2)) (Const 23)=42
")
```

Monads

What is a monad?

Common patterns

1. for each type `a` of *values*, a type `M a` represents *computations*
   - `eval :: Exp -> Int` becomes `evalX :: Exp -> M Int`
   - *in general* `a -> b` becomes `a -> M b`
2. for `Const i` we need to turn values into computations

   ```
   return  :: a -> M a
   ```

3. for `` e1 `div` e2 `` we need to combine computations

   ```
   >>= :: M a -> (a -> M b) -> M b
   ```

Have you seen »= before?

Monads as a type class

Monad type class

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

    fail        :: String -> m a

    m >> k      = m >>= \_ -> k
    fail s      = error s
```

monad laws

```
return a >>= k  ==  k a
m >>= return  ==  m
m >>= (\x -> k x >>= h)  ==  (m >>= k) >>= h
```

Monads and functors

Alternative definition of `Monad`

- `return` is `pure`
- but this is not true in HASKELL (yet!)
  (coming to GHC near you with 7.10)

```
class Applicative m => Monad' m where
  (>>=) :: m a -> (a -> m b) -> m b
```

Extra law

Instances of both `Monad` and `Functor` should additionally satisfy the law:

```
fmap f xs  ==  xs >>= return . f
```

Monads as computations

- monads
  - are a way to *structure computations* in terms of *values* and *sequences of computations* using those values
  - allow the programmer to *build* up computations using *sequential building blocks*
  - determine how *combined computations* form a new computation
- roughly speaking:
  - *type constructor* defines a type of computation
  - `return` creates primitive values of computation
  - `>>=` (*bind*) combines computations together
- *container analogy*

Monadic evaluators

Monadic evaluator

the identity monad

```
data Id t = Id t deriving Show
instance Monad Id where
   return x = Id x
   (Id x) >>= f = f x
```

monadic evaluator

```
evalM :: Exp -> Id Int
evalM (Div e1 e2) = evalM e1 >>= (\a ->
                    evalM e2 >>= (\b ->
                    return (a `div` b)))
evalM (Const i)   = return i
```

Monadic error handling

the monad:

```
data ME a = Error Exception | Ok a deriving Show

instance Monad ME where
   return a = Ok a
   m >>= f  = case m of Error s  -> Error s
                        Ok a -> f a

raise :: Exception -> ME a
raise s = Error s
```

the evaluator:

```
evalME0 :: Exp -> ME Int
evalME0 (Div e1 e2) = evalME0 e1 >>= (\a ->
                      evalME0 e2 >>= (\b ->
                      if b == 0 then (raise "division by 0")
                                else return (a `div` b)))
evalME0 (Const i)   = return i
```

do notation 1/2

The code for `evalME0 (Div e1 e2)` is tedious:

```
evalME0 (Div e1 e2) = evalME0 e1 >>= (\a ->
                      evalME0 e2 >>= (\b ->
                      if b == 0 then (raise "division by 0")
                                else return (a `div` b)))
evalME0 (Const i)   = return i
```

it would be much nicer to write code like this:

```
evalME (Div e1 e2) = do a <- evalME e1;
                        b <- evalME e2;
                        if b == 0 then (raise "division by 0")
                          else return (a `div` b)
evalME (Const i)   = return i
```

- the latter approach is called the *do notation*
- one of the advantages of membership in the `Monad` class

do notation 2/2

desugaring the do blocks

`do` e	$\longrightarrow$	e
`do {`e`;` $stmts$`}`	$\longrightarrow$	e `>> do {`$stmts$`}`
`do {`v `<-` e`;` $stmts$`}`	$\longrightarrow$	e `>>=\`v `-> do {`$stmts$`}`
`do {let` $decls$`;` $stmts$`}`	$\longrightarrow$	`let` $decls$ `in do {`$stmts$`}`

- very “imperative” feel
- the desugaring is *almost* like this
- special case: *v* is a *pattern*, not a variable:

```
do (x:xs) <- foo
   bar x
```

- if `foo` is `[]`, then `fail` is called
- see the *Haskell Report* for details

Monadic state

the monad

```
newtype MS a = MS { runMS :: (State -> (a, State)) }

instance Monad MS where
   return a  = MS (\x -> (a, x))
   m >>= f   = MS $ \x -> let (a, y) = runMS m x in
                          let (b, z) = runMS (f a) y in
                          (b, z)


tick = MS (\x -> ((),x+1))
```

the evaluator

```
evalMS :: Exp -> MS Int
evalMS (Div e1 e2) = do a <- evalMS e1;
                        b <- evalMS e2;
                        tick;
                        return (a `div` b)
evalMS (Const i)   = return i
```

Monadic output

the monad

```
newtype MO a = MO (a, Output) deriving Show

instance Monad MO where
   return a = MO (a, "")
   m >>= f  = MO $ let MO (a,x) = m in
                   let MO (b,y) = f a in
                   (b, x ++ y)

out :: Output -> MO ()
out s = MO((), s)
```

the evaluator

```
evalMO :: Exp -> MO Int
evalMO (Div e1 e2) = do a <- evalMO e1;
                        b <- evalMO e2;
                        out (line (Div e1 e2) (a `div` b));
                        return (a `div` b)
evalMO (Const i)   = do out (line (Const i) i);
                        return i
```

Monad lifting

- to make “ordinary” functions operate on monadic values

```
liftM   :: (Monad m) => (a1 -> r) -> m a1 -> m r
liftM f m1       = do { x1 <- m1; return (f x1) }
```

- example

```
> liftM (*3) (Just 8)
Just 24
```

- also for functions of two arguments:

```
liftM2  :: (Monad m) => (a1 -> a2 -> r) -> m a1 -> m a2 -> m r
liftM2 f m1 m2   = do { x1 <- m1; x2 <- m2; return (f x1 x2) }
```

- and all the way to `liftM5`

Meet the Monads

The `Identity` monad

```
newtype Identity a = Identity { runIdentity :: a }

instance Monad Identity where
    return a = Identity a
    m >>= k  = k (runIdentity m)
```

- no computational strategy
- just applies functions to arguments
- raison d'etre: monad transformers

The `Maybe` monad

```
data  Maybe a  =  Nothing | Just a

instance  Monad Maybe  where
    (Just x) >>= k      = k x
    Nothing  >>= _      = Nothing
    return              = Just
    fail _              = Nothing
```

- for combining a chain of computations
- each can fail (`Nothing`) or produce a result (`Just x`)
- once one fails, the whole chain fails too
- without it:

```
case ... of
  Nothing -> Nothing
  Just x  -> case ... of
               Nothing -> Nothing
               Just y  -> ...
```

The Error monad

```
class Error a where
  noMsg :: a
  strMsg :: String -> a

class Monad m => MonadError e m | m -> e where
  throwError :: e -> m a
  catchError :: m a -> (e -> m a) -> m a

instance MonadError e (Either e) where
    throwError             = Left
    Left  l `catchError` h = h l
    Right r `catchError` _ = Right r
```

- facilitates *exception handling*
- typical use:

```
do { action1; action2; action3 } `catchError` handler
```

Evaluator using the `Error` monad

```
type MLE = Either String

evalMLE :: Exp -> MLE Int
evalMLE (Div e1 e2) = do
               a <- evalMLE e1;
               b <- evalMLE e2;
               if b == 0 then (throwError "division by 0")
                         else return (a `div` b)
evalMLE (Const i)   = return i
```

- this version uses `Either String`
- we could have used custom error type

The `State` monad

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

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

- values are transition functions from an initial state to a (value, newState) pair

The `MonadState` class

```
class (Monad m) => MonadState s m | m -> s where
    get :: m s
    put :: s -> m ()
```

- provides a simple interface for State monads
- `get` retrieves a state by copying it into the value
- `put` sets the state, but does not yield a value

```
instance MonadState s (State s) where
    get   = State $ \s -> (s, s)
    put s = State $ \_ -> ((), s)
```

Evaluator using the State monad

```
type MLS = State Int

tick :: (Num s, MonadState s m) => m ()
tick = do st <- get;
          put (st+1)

evalMLS :: Exp -> MLS Int
evalMLS (Div e1 e2) = do a <- evalMLS e1;
                         b <- evalMLS e2;
                         tick;
                         return (a `div` b)
evalMLS (Const i)   = return i

runMLS s exp = runState (evalMLS exp) s
```

```
> runMLS 0 answer
(42,2)
```

The `Reader` monad

```
newtype Reader r a = Reader { runReader :: r -> a }

instance Monad (Reader r) where
    return a = Reader $ \_ -> a
    m >>= k  = Reader $ \r -> runReader (k (runReader m r)) r
```

- provides read-only environment (e.g. variable bindings)
- for some applications better suited than `State` (clearer, easier)
- `return` ignores the environment

The `MonadReader` class

```
class (Monad m) => MonadReader r m | m -> r where
    ask   :: m r
    local :: (r -> r) -> m a -> m a

instance MonadReader r (Reader r) where
    ask       = Reader id
    local f m = Reader $ runReader m . f

asks :: (MonadReader r m) => (r -> a) -> m a
asks f = do  r <- ask;
             return (f r)
```

- convenience functions for the `Reader` monad
    - `ask` retrieves the environment
    - `local` executes computation in modified environment
    - `asks` retrieves function of the current environment

Other monads

List	multiple values of nondeterministic computation
IO	the “magic” IO monad
Writer	produces stream of data in addition to the computed values
Continuation	for continuations

Reading

- P. Wadler: *Monads for functional programming.* Marktoberdorf 1992.
- B. Yorgey: *The Typeclassopedia* `https://www.haskell.org/haskellwiki/Typeclassopedia`
- *All About Monads* `https://www.haskell.org/haskellwiki/All_About_Monads`
- D. Piponi: *You Could Have Invented Monads! (And Maybe You Already Have.)* `http://blog.sigfpe.com/2006/08/you-could-have-invented-monads-and.html`