EDAF95/EDAN40: Functional Programming Functors and Monads

Jacek Malec
Dept. of Computer Science, Lund University, Sweden

April 5th, 2023

Monad class

Motivation:

- **Separation of *pure* and sequential code**
- Properties of a particular kind of functions
- Introduction of *state* and its transformations
- Or simply: yet another type class

For more material: see the course web.

The functional IO problem

IO has traditionally been included as side effects to ordinary functions:

```
inputInt :: Int
```

then substituting equals for equals is no longer possible

```
inputDiff = inputInt - inputInt
```

probably not equal to zero

Example: Lambda laughter

Suppose the function:

```
outputChar :: char -> ()
```

has the side effect of printing its argument.
The expression (note: pseudo-Haskell):

```
outputChar 'h'; outputChar 'a';
    outputChar 'h'; outputChar 'a'
```

Should then print the string `"haha"`.

Lambda laughter, cont.

If we then try to catch the repetitive pattern by instead writing:

```
let x = (outputChar 'h'; outputChar 'a') in
    x; x
```

“then the laugh is on us” (P. Wadler). It will print only `"ha"`. Why?

Lambda laughter, cont.

If we then try to catch the repetitive pattern by instead writing:

```
let x = (outputChar 'h'; outputChar 'a') in
    x; x
```

"then the laugh is on us" (P. Wadler). It will print only `"ha"`. Why?

However, the following (note: pseudo-Haskell):

```
let f() =  (outputChar 'h'; outputChar 'a') in
    f(); f()
```

will print the string `"haha"`.

The IO Monad

The type of IO **activities**:

```
prompt :: IO ()
prompt = putStr ">:"
```

The type of IO activities that return a value:

```
getChar :: IO Char
getLine :: IO String
```

The do-notation

Allows sequencing and naming the returned values:

```
echoReverse :: IO ()
echoReverse = do
    aLine <- getLine
    putStrLn (reverse aLine)
```

return and let

The *return* operation does the empty IO-activity:

```
getInt :: IO Int
getInt = do
    aLine <- getLine
    return (read aLine :: Int)
```

Local variables may be defined using *let*:

```
echoReverse2 :: IO ()
echoReverse2 = do
    aLine <- getLine
    let theLineReversed = reverse aLine
    putStrLn (theLineReversed)
```

Imperative style

One can e.g. use the conditional clause:

```
testPalindrome :: IO ()
testPalindrome = do
    prompt
    aLine <- getLine
    if (aLine == (reverse aLine)) then
        putStrLn "Yes, a palindrome."
      else
        putStrLn "No, no!"
```

Loops done with recursion

```
testPalindroms :: IO ()
testPalindroms = do
   prompt
   aLine <- getLine
      if (aLine == "") then
          return ()
        else do
          if (aLine == (reverse aLine)) then
              putStrLn "Yes, a palindrome."
            else
              putStrLn "No, no!"
          testPalindroms
```

Monadic lambda laughter

The IO monad gives type-safe laughters:

```
laugh :: IO ()
laugh =
    let x = do putChar 'h'; putChar 'a'
    in do x; x
```

IO-stripping

IO stripping is not allowed (at least officially:-) because if you could strip off side effects with:

```
stripIO :: IO a -> a
```

then the following code

```
inputInt :: Int
inputInt = ioStrip getInt

inputDiff = inputInt - inputInt
```

would violate “equals for equals” principle.

Back door

In the module `System.IO.Unsafe` there actually is

```
unsafePerformIO :: IO a -> a
```

behaving as expected. However:

- it is **NOT** type-safe;
- you could coerce any type to any other type using `unsafePerformIO`!

FORGET IT IMMEDIATELY!

Monad class (again)

Motivation:

- Separation of *pure* and sequential code
- **Properties of a particular kind of functions**
- Introduction of *state* and its transformations
- Or simply: yet another type class

Composing functions

Composing functions is simple:

```
f :: a -> b
g :: b -> c
```

then `g` and `f` may be composed to `g.f`.

But suppose:

```
f :: a -> Maybe b
g :: b -> Maybe c
```

then how to compose `g` and `f`? And what will it mean?

Composing functions

Yet another example:

```
children      :: Person -> [Person]
grandchildren :: Person -> [Person]
```

then almost

```
grandchildren = children.children
```

but not exactly.

Functor (reminder)

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

Please note that

```
instance Functor IO where
    fmap f action = do
        result <- action
        return (f result)
```

More functors

An example of its utility:

```
import Data.Char
import Data.List
fmaptest = do
  line <- fmap (intersperse '-' . reverse . map toUpper)
                                                  getLine
  putStrLn line
```

More functors

An example of its utility:

```
import Data.Char
import Data.List
fmaptest = do
  line <- fmap (intersperse '-' . reverse . map toUpper)
                                                  getLine

  putStrLn line
```

Slightly more complicated:

```
ghci> let a = fmap (*) [1,2,3,4]
ghci> :t a
a :: [Integer -> Integer]
ghci> fmap (\f -> f 9) a
[9,18,27,36]
```

Applicative functors

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

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

Applicative functors: examples

```
ghci> Just (+3) <*> Just 9
Just 12
ghci> pure (+3) <*> Just 10
Just 13
ghci> pure (+3) <*> Just 9
Just 12
ghci> Just (++"hahah") <*> Nothing
Nothing
ghci> Nothing <*> Just "woot"
Nothing
ghci> pure (+) <*> Just 3 <*> Just 5
Just 8
ghci> pure (+) <*> Just 3 <*> Nothing
Nothing
ghci> pure (+) <*> Nothing <*> Just 5
Nothing
```

Some more ...

So we can apply:

```
pure fn <*> x <*> y <*> ...
```

Considering that `pure fn <*> x` equals `fmap fn x` we can write instead

```
fmap fn x <*> y <*> ...
```

or even more cleanly:

```
(<$>) :: (Functor f) => (a -> b) -> f a -> f b
fn <$> x = fmap fn x

fn <$> x <*> y <*> ...

ghci> [(+),(*)] <*> [1,2] <*> [3,4]
[4,5,5,6,3,4,6,8]
```

Rounding up

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

instance Applicative IO where
    pure = return
    a <*> b = do
        f <- a
        x <- b
        return (f x)
```

The Monad 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

-- Minimal complete definition:
--       (>>=), return
    m >> k  =  m >>= \_ -> k
    fail s  = error s
```

Operations >>= (bind) and >> are (by some) pronounced “then”.

Just a word on return, again

`return` is a bad name!

```
main = do
  s <- getLine
  return ()
  putStrLn s
```

works as charm!

The MonadPlus class

```
class (Monad m) => MonadPlus m where
    mzero :: m a
    mplus :: m a -> m a -> m a
```

Algebraically: a monoid

The identity monad

```
data Id a = Id a

instance Monad Id where
   return x =   Id x
   (Id x)   >>= f  = f x
```

The List monad

```
instance Monad [] where
    return x = [x]
    xs >>= f = concat (map f xs)
    fail s   = []

instance MonadPlus [ ] where
    mzero = []
    mplus = (++)
```

The Maybe monad

```
instance Monad Maybe where
    return x       = Just x
    Just x  >>= f = f x
    Nothing >>= f = Nothing

instance MonadPlus Maybe where
    mzero                 = Nothing
    Nothing `mplus` ys = ys
    xs `mplus` ys      = xs
```

The do-notation

do-expressions are just syntactic sugar for >>= or >>:

```
echoReverse = do
    aLine <- getLine
    putStrLn (reverse aLine)
```

is just

```
echoReverse =
    getLine >>= \aLine ->
    putStrLine (reverse aLine)
```

List comprehensions

"Reverse" do-notation:

```
list1 = [ (x,y) | x<-[1..], y<-[1..x]]

list2 = do
    x <- [1..]
    y <- [1..x]
    return (x,y)
```

Monad class

Motivation:

- Separation of *pure* and sequential code
- Properties of a particular kind of functions
- **Introduction of *state* and its transformations**
- Or simply: yet another type class

Monadic Modularity

A simple random number generator:

```
g :: Integer -> (Float, Integer)
g seed = (fromInteger(newSeed)/fromInteger(m), newSeed)
   where
      newSeed = (seed*a) `mod` m
      a = 1812433253
      m = 2^32

initialSeed = 3121281023
```

Usage

```
(a, s1) = g initialSeed
(b, s2) = g s1
(c, s3) = g s2

(xs, s4) = (values, last seeds)
   where (values, seeds) =
      (unzip.take 17.tail) (iterate (g.snd) (dummy, s3))
      dummy = 45.0
```

Quite clumsy.

A generalisation attempt

```
newtype RandomGenerator a = Ran (Integer -> (a, Integer))

random = Ran g
generate (Ran f) = (fst.f) initialSeed
```

The Random monad

```
type R = RandomGenerator

instance Monad RandomGenerator where
    return :: a -> R a
    return x = Ran (\seed -> (x, seed))

    (>>=) :: R a -> (a -> R b) -> R b
    (Ran g0) >>= f = Ran (\seed ->
        let (y, seed1) = g0 seed
            (Ran g1) = f y
        in g1 seed1)
```

Using the random monad

```
randoms3 = do
   a <- random
   b <- random
   c <- random
   return (a,b,c)

result3 = generate randoms3

randomList 0 = return []
randomList n = do
   x  <- random
   xs <- randomList (n-1)
   return (x:xs)
```

do-notation, again

```
randomPair = do
    a <- random
    b <- random
    return (a,b)
```

is equivalent to

```
randomPair =
    random >>= \a ->
    random >>= \b ->
    return (a,b)
```

or

```
randomPair = random >>= (\a -> random >>= (\b -> return (a,b)))
```

The pattern

State transformation and value generation

Another state transformation example

```
type Dictionary = [(String,String)]

dictAdd key value dict = (key, value):dict
dictFind key dict = lookup key dict
```

Passing state without side effects is clumsy:

```
result1 = r where
   d1 = dictAdd  "no" "norway" []
   d2 = dictAdd  "se" "sweden" d1
   r1 = dictFind "fr" d2
   d3 = dictAdd "fr" "france" d2
   r  = dictFind "fr" d3
```

The StateTransform monad

```
newtype StateTransform s a = ST (s -> (s,a))
apply (ST f) = f

instance Monad (StateTransform s) where
   return x  = ST $ \s -> (s, x)
   x >>= f   = ST $ \s0 ->
                    let (s1,y) = apply x s0
                    in (apply (f y)) s1

stateUpdate :: (s -> s) -> StateTransform s ()
stateUpdate u = ST $ \s -> (u s, ())
stateQuery :: (s -> a) -> StateTransform s a
stateQuery q = ST $ \s -> (s, q s)
runST s t = apply t s
```

Dictionary example again

```
type DictMonad = StateTransform Dictionary

result2 = snd (runST [] dictM) where
   dictM :: DictMonad (Maybe String)
   dictM = do
      stateUpdate (dictAdd "no" "norway")
      stateUpdate (dictAdd "se" "sweden")
      r1 <- stateQuery (dictFind "fr")
      stateUpdate (dictAdd "fr" "france")
      r <- stateQuery (dictFind "fr")
```

State passing becomes invisible: robustness.

Monad class

Motivation:

- Separation of *pure* and sequential code
- Properties of a particular kind of functions
- Introduction of *state* and its transformations
- Or simply: **yet another type class**

Re: definition

Kleisli triple

1. A **type construction**: for type *a* create *Ma*
2. A **unit function** $a \rightarrow Ma$ (`return` in Haskell)
3. A **binding operation** of polymorfic type $Ma \rightarrow (a \rightarrow Mb) \rightarrow Mb$. Four stages (informally):
   1. The monad-related structure on the first argument is "pierced" to expose any number of values in the underlying type a.
   2. The given function is applied to all of those values to obtain values of type (M b).
   3. The monad-related structure on those values is also pierced, exposing values of type b.
   4. Finally, the monad-related structure is reassembled over all of the results, giving a single value of type (M b).

Re: definition

Monad axioms:

1. *return* acts as a neutral element of $>>=$.

$$(return\ x) >>= f \Leftrightarrow f\ x$$

$$m >>= return \Leftrightarrow m$$

2. Binding two functions in succession is the same as binding one function that can be determined from them.

$$(m >>= f) >>= g \Leftrightarrow m >>= \lambda x.(f\ x >>= g)$$

Re: definition

We can restate it in a cleaner way (Thomson). Define

```
(>@>) :: Monad m => (a -> m b) -> (b -> m c) -> (a -> m c)

f >@> g = \x -> (f x) >>= g
```

Now, the monad axioms may be written as:

```
return >@> f = f
f >@> return = f

(f >@> g) >@> h = f >@> (g >@> h)
```

Example

```
a = do x <- [3..4]
       [1..2]
       return (x, 42)
```

is equivalent to

```
a = [3..4] >>= (\x -> [1..2] >>= (\_ -> return (x, 42)))
```

Thus, remembering that

```
instance Monad [] where
  m >>= f  = concatMap f m
  return x = [x]
  fail s   = []
```

Example, cont.

The transformations may be reduced as follows:

```
a = [3..4] >>= (\x -> [1..2] >>= (\_ -> return (x, 42)))
a = [3..4] >>=
              (\x -> concatMap (\_ -> return (x, 42)) [1..2])
a = [3..4] >>= (\x -> [(x,42),(x,42)] )
a = concatMap (\x -> [(x,42),(x,42)] ) [3..4]
a = [(3,42),(3,42),(4,42),(4,42)]
```