Class 3: Monads,* v1.2

Jacek Malec (but see Introduction)

November 28, 2013

1 Introduction

This document is a derivative of many sources, most notably (in no particular order):

1. "Haskell" on WikiBooks: `http://en.wikibooks.org/wiki/Haskell/`;
2. Bryan O'Sullivan, John Goerzen and Don Stewart, "Real World Haskell", O'Reilly, 2009;
3. Simon Thompson, "The Craft of Functional Programming", 2nd ed., (formally our course book), Addison-Wesley, 1999;
4. Bernie Pope, "A tour of the Haskell Prelude", 2001;
5. Materials of TDA555 from Chalmers, `http://www.cse.chalmers.se/edu/course/TDA555/`.

2 Exercises

2.1 Simple definitions (Thompson)

Show that sets and binary trees can be given a monad structure. Show the same for the type

```
data Error a = OK a | Error String
```

2.2 Monadic helper functions (TDA555)

Give an implementation of the following two functions:

```
sequence_ :: Monad m => [m ()] -> m ()
onlyIf    :: Monad m => Bool -> m () -> m ()
```

`sequence_` takes a list of instructions resulting in an uninteresting value, and creates one big instruction that executes all of these.
`onlyIf` takes a boolean and an instruction, and creates an instruction that only executes the argument instruction if the boolean was `True`. If the boolean

*Intended for EDAN40 course, after the monad lecture.

was `False`, nothing happens. Example `onlyIf failed tryAgain` executes the instructions `tryAgain` only if the boolean `failed` is `True`.
Hint: You might find it easier to think of the above functions having type:

```
sequence_ :: [IO ()] -> IO ()
onlyIf    :: Bool -> IO () -> IO ()
```

What becomes different if we change the type of `onlyIf` to:

```
onlyIfM :: Monad m => m Bool -> m () -> m ()
```

What other kinds of programs can we write now?
Give an implementation of `onlyIfM`.

2.3 List comprehension (lecture)

Let us take the example from the lecture:

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

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

Rewrite `list2` using bind (`>>=`) instead of `do`.

2.4 Some theorem proving (Thompson)

First, let us remind that the full Monad class is defined as follows:

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

where `(>>)` and `fail` may be defined as defaults:

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

So `>>` just discards the value of the first expression, whereas `>>=` passes it to the next one.

Let us now define yet another derived operator, `>@>`:

```
(>@>) :: Monad m => (a -> m b) ->
                    (b -> m c) ->
                    (a -> mc)
f >@> g = \x -> (f x) >>= g
```

This is the *Kleisli composition*.

Let us now formulate three requirements on the operations of a monad. First, `return` acts as identity for the operator `>@>`:

```
return >@> f = f                      (M1)
f >@> return = f                      (M2)
```

and the operator `>@>` should be associative:

```
(f >@> g) >@> h = f >@> (g >@> h)     (M3)
```

Task 1: For the monads `Id`, `[]` and `Maybe` prove the rules `(M1)` to `(M3)`.

Let us define some standard functions over every monad:

```
fmap :: Monad m => (a -> b) -> m a -> m b
join :: Monad m => m (m a) -> m a

fmap f m
  = do x <- m
       return (f x)

join m
  = do x <- m
       x
```

Over the lists they correspond to `map` and `concat`, respectively. Now we can formulate the following dependency:

```
fmap (f . g) = fmap f . fmap g            (M4)
```

Task 2: Prove the property `(M4)` using the laws `(M1)` to `(M3)`.

Task 3: Prove the following properties using the monad laws:

```
join return = join . fmap return
join return = id
```

Task 4: Write down the definitions of `map` and `join` over lists using list comprehensions. Compare them with the definitions of `fmap` and `join` given in the do-notation above.

2.5 State Monad (RealWorldHaskell)

You may wish to have a look at Chapter 14 of "Real World Haskell"[1] where, among others, the State monad is gently introduced. In particular, the following set of definitions come there:

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

bindState :: State s a -> (a -> State s b) -> State s b
bindState m k = State $ \s -> let (a, s') = runState m s
                              in runState (k a) s'
```

[1] `http://book.realworldhaskell.org`

```
instance Monad (State s) where
    return a = returnState a
    m >>= k = bindState m k

get :: State s s
get = State $ \s -> (s, s)

put :: s -> State s ()
put s = State $ \_ -> ((), s)
```

Task 1: (and only one:-) Rewrite the random number generator as an instance of the state monad, using the functions `put` and `get`.

The `StateTransform` monad from the lecture notes is actually equivalent to the above `State` monad, with possibly a slightly different set of help functions defined, in particular:

```
updateST :: (s -> s) -> State s ()
updateST u = State $ \s -> ((), u s)

queryST   :: (s -> a) -> State s a
queryST q = State $ \s -> (q s, s)
```

Finally, please note that the `StateTransform` monad is **not** a monad transformer (as defined in Chapter 18 of RWH).