Control Monad (9A)

Young Won Lim
9/8/18

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice.

Based on

Haskell in 5 steps

https://wiki.haskell.org/Haskell_in_5_steps

sequence

sequence is a **function** which takes
a list of computations of the same type,
and builds from them a computation
which will run each in turn and
produce a list of the results:

```
sequence :: (Monad m) => [m a] -> m [a]
sequence []     = return []
sequence (x:xs) = do v <- x
                     vs <- sequence xs
                     return (v:vs)
```

https://wiki.haskell.org/Monads_as_computation

sequence

```
sequence :: (Monad m) => [m a] -> m [a]
sequence []     = return []
sequence (x:xs) = do v <- x
                     vs <- sequence xs
                     return (v:vs)

sequence :: (Monad m) => [m a] -> m [a]
sequence []     = return []
sequence (x:xs) = x >>= \v -> sequence xs >>= \vs -> return (v:vs)
```

https://wiki.haskell.org/Monads_as_computation

sequence

sequence :: Monad m => [m a] -> m [a]

evaluate **each action** in the sequence from left to right,

and collect the **results**.

sequence [(> 4), (< 10), odd] 7

[True, True, True]

sequence [fmap (*2) , fmap (*3) , fmap (*4)] (Just 2)

[Just 4,Just 6,Just 8]

sequence [((*2) <$>) , ((*3) <$>) , ((*4) <$>)] (Just 2)

[Just 4,Just 6,Just 8]

http://derekwyatt.org/2012/01/25/haskell-sequence-over-functions-explained/

Sequence

```
sequence :: (Monad m) => [m a] -> m [a]
sequence []     = return []
sequence (x:xs) = do v <- x
                     vs <- sequence xs
                     return (v:vs)
```

without the do-notation:

```
sequence :: (Monad m) => [m a] -> m [a]
sequence []     = return []
sequence (x:xs) = x >>= \v -> sequence xs >>= \vs -> return (v:vs)
```

(one can start to see why do-notation might be desirable!)

It's a function which takes a list of computations of the same type, and builds from them a computation which will run each in turn and produce a list of the results:

http://derekwyatt.org/2012/01/25/haskell-sequence-over-functions-explained/

sequence

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

...while sequence can be defined like this:

```
sequence :: [m a] -> m [a]
sequence [] = return []
sequence (m:ms) = m >>= (\x -> sequence ms >>= (\xs -> return $ x:xs))
```

https://www.reddit.com/r/haskellquestions/comments/6xk5hv/the_sequence_function/

sequence

In a parsing monad, we might pass it a list of parsers,

and get back a parser which parses its input using each in turn.

In the IO monad, a simple example might be the following:

main = sequence [getLine, getLine] >>= print

which gets two lines of text from the user,

and then prints the list.

Since lists are lazy in Haskell,

this gives us a sort of primordial loop

from which most other kinds of loops can be built.

https://wiki.haskell.org/Monads_as_computation

forM

a for-each loop is something which performs some action
based on each element of a list.
think a function with the type:
collect the results of each iteration
We can write this with sequence and map:

```
forM :: (Monad m) => [a] -> (a -> m b) -> m [b]
forM xs f = sequence (map f xs)
```

we apply the function to each element of the list
to construct the action for that iteration,
and then sequence the actions together into a single computation.

https://wiki.haskell.org/Monads_as_computation

forM

```
forM :: (Monad m) => [a] -> (a -> m b) -> m [b]
forM xs f = sequence (map f xs)

main = forM [1..10] $ \x -> do
          putStr "Looping: "
          print x
```

Since in this, and many other cases,

the loop body doesn't produce a particularly interesting result,

there are variants of **sequence** and **forM**

called **sequence**_and **forM**_

which simply throw the results away as they run each of the actions.

https://wiki.haskell.org/Monads_as_computation

Sequence_, forM_

```
sequence_ :: (Monad m) => [m a] -> m ()
sequence_ []     = return ()
sequence_ (x:xs) = x >> sequence_ xs

forM_ :: (Monad m) => [a] -> (a -> m b) -> m ()
forM_ xs f = sequence_ (map f xs)
```

https://wiki.haskell.org/Monads_as_computation

when

Sometimes we only want a computation to happen

when a given condition is true.

when :: (Monad m) => Bool -> m () -> m ()

when p x = if p then x else return ()

Remember that **return ()** is a no-op,

so running this computation will run x

when the condition is true,

and will do nothing at all when the condition fails.

https://wiki.haskell.org/Monads_as_computation

liftM

Another extremely common thing to do is

to construct a computation which performs another computation and

then applies a function to the result.

This can be accomplished by using the **liftM** function:

```
liftM :: (Monad m) => (a -> b) -> m a -> m b
liftM f x = do v <- x
               return (f v)
```

```
liftM :: (Monad m) => (a -> b) -> m a -> m b
liftM f x = return . f =<< x
```

Where **(=<<)** is just bind with its parameters flipped.

https://wiki.haskell.org/Monads_as_computation

liftM2

This is also generalised by liftM2, liftM3, ...

to running more than one computation

before applying a function to the results:

```
liftM2 :: (Monad m) => (a -> b -> c) -> m a -> m b -> m c
liftM2 f x y = do v <- x
                  w <- y
                  return (f v w)
```

https://wiki.haskell.org/Monads_as_computation

sequence via liftM2, return, fold

It's possible to rewrite **sequence**

in terms of **liftM2**, **return**, and a **fold** over the list:

```
sequence :: (Monad m) => [m a] -> m [a]
sequence xs = foldr (liftM2 (:)) (return []) xs

sequence_ :: (Monad m) => [m a] -> m ()
sequence_ xs = foldr (>>) (return ()) xs
```

Anyway, these are just a few of the simpler examples

to give a taste of what sorts of control structures you get for free

by defining a combinator library as a monad.

https://wiki.haskell.org/Monads_as_computation

References

[1] ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf

[2] https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf