State Monad (3E)

Young Won Lim
10/7/17

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

This document was produced by using OpenOffice.

Based on

Haskell in 5 steps
https://wiki.haskell.org/Haskell_in_5_steps

Maybe Monad

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




instance Monad Maybe where
   -- return      :: a -> Maybe a
   return x       =  Just x

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

```
f :: a -> m b
```

https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

Maybe Monad

a monad is a parameterized type **m**
that supports **return** and **>>=** functions of the specified types

m must be a parameterized type,
rather than just a type (not a concrete type)

It is because of this declaration
that the **do** notation can be used to sequence **Maybe** values.

More generally, Haskell supports the use of this notation
with any monadic type.

examples of types that are monadic,
the benefits that result from recognizing and exploiting this fact.

https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

Monad, Monoid

monad (plural monads)

- An ultimate atom, or simple, unextended point; something ultimate and indivisible.
- (mathematics, computing) A monoid in the category of endofunctors.
- (botany) A single individual (such as a pollen grain) that is free from others, not united in a group.

monoid (plural monoids)

- (mathematics) A **set** which is closed under an associative binary operation, and which contains an element which is an identity for the operation.

https://en.wiktionary.org/wiki/monad, monoid

List Monad

The **Maybe** monad provides a simple model of computations that can fail,

a value of type Maybe a is either Nothing (failure)

the form Just x for some x of type a (success)

The **list** monad generalises this notion,

by permitting multiple results in the case of success.

More precisely, a value of [a] is

either the empty list [] (failure)

or the form of a non-empty list [x1,x2,...,xn] (success)

for some xi of type a

https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

List Monad

```
instance Monad [] where
   -- return :: a -> [a]
   return x  =  [x]

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

return converts a value into a *successful* result containing that value

>>= provides a means of *sequencing* computations
that may produce *multiple results*:

xs >>= **f** applies the function **f** to each of the *results* in the list **xs**
to give a *nested list* of *results*,
which is then concatenated to give a *single list* of *results*.

(Aside: in this context, [] denotes the list type [a] without its parameter.)

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

https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

List Monad

```
instance Monad [] where
  -- return :: a -> [a]
  return x  =  [x]

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

```
instance Monad ST where
  -- return :: a -> ST a
  return x  =  \s -> (x,s)

  -- (>>=)  :: ST a -> (a -> ST b) -> ST b
  st >>= f  =  \s -> let (x,s') = st s in f x s'
```

https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

A State Transformer

```
type State = ...

type ST = State -> State
```

the problem of writing functions that manipulate some kind of state, represented by a type, whose detail is not our concern now.

a state transformer (ST), which takes the current state as its argument, and produces a modified state as its result, which reflects any side effects performed by the function:

https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

A Generalized State Transformer

type State = ...

type ST = State -> State

type ST a = State -> (a, State)

want to return a result value in addition to the modified state
generalized state transformers also return a result value,
as a parameter of the ST type

State -> (a, State)
s -> (v, s')

s: input state, v: the result value, s': output state

https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

A Curried Generalized State Transformer

```
type ST a = State -> (a, State)                    generalized ST

type ST' a b  = b -> State -> (a, State)           further generalized ST

        b -> ST a = b -> State -> (a, State)       think currying
```

also may need to take argument values
no need to use more generalized ST type
can be exploiting currying.

a state transformer that takes a character and returns an integer
would have type Char -> ST Int

Char -> State -> (Int, State) **curried form**

https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

ST Monad

```
instance Monad ST where
  -- return :: a -> ST a
  return x  =  \s -> (x,s)

  -- (>>=)  :: ST a -> (a -> ST b) -> ST b
  st >>= f  =  \s -> let (x,s') = st s in f x s'
```

>>= provides a means of sequencing state transformers:
st >>= **f** applies the state transformer **st** to an initial state s,
then applies the function **f** to the resulting value x
to give a second state transformer (**f** x),
which is then applied to the modified state s' to give the final result:

st :: ST a
f :: a -> ST b
(>>=) :: ST a -> (a -> ST b) -> ST b

st :: State -> (a, State)
f :: a -> State -> (b, State)
(>>=) :: State -> (a, State) -> (a -> ST b)
-> ST b

(x,s') = **st** s

f x s'

https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

ST Monad

```
instance Monad ST where
   -- return :: a -> ST a
   return x  =  \s -> (x,s)

   -- (>>=)  :: ST a -> (a -> ST b) -> ST b
   st >>= f  =  \s -> let (x,s') = st s in f x s'

st :: ST a
f :: a -> ST b
(>>=) :: ST a -> (a -> ST b) -> ST b

st :: State -> (a, State)            (x,s') = st s    s → (x,s')
f :: a -> State -> (b, State)         f x s'
(>>=) :: State -> (a, State) -> (a -> State -> (b, State)) -> State -> (b, State)
```

https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

let … in …

```
cylinder :: (RealFloat a) => a -> a -> a
cylinder r h =
    let sideArea = 2 * pi * r * h
        topArea = pi * r ^2
    in  sideArea + 2 * topArea
```

The form is let <bindings> in <expression>.

The names that you define in the let part
are accessible to the expression after the in part.

Notice that the names are also aligned in a single column.

For now it just seems that let puts the bindings first
and the expression that uses them later
whereas where is the other way around.

http://learnyouahaskell.com/syntax-in-functions

List Monad

```
instance Monad [] where
  -- return :: a -> [a]
  return x  =  [x]

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

```
instance Monad Maybe where
  -- return      :: a -> Maybe a
  return x       =  Just x

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

```
instance Monad ST where
  -- return :: a -> ST a
  return x  =  \s -> (x,s)

  -- (>>=)  :: ST a -> (a -> ST b) -> ST b
  st >>= f  =  \s -> let (x,s') = st s in f x s'
```

https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

Dummy Constructor S0

```
type ST a = State -> (a, State)          generalized ST

data ST0 a = S0 (State -> (a, State))
```

types defined using the **type** mechanism
cannot be made into instances of classes.

types defined using the **data** mechanism
can be made into instances of classes.
but requires a dummy constructor (S0)

https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

Removing Data Constructor

```
type ST a = State -> (a, State)                    generalized ST

data ST0 a = S0 (State -> (a, State))
```

In order to remove the dummy constructor,
define our own application function (**apply0**) for this type

```
apply0  :: ST0 a -> State -> (a, State)

apply0 (S0 f) x = f x

f  :: State -> (a, State)
```

*** Curried Function**

f x y	**f :: a -> b -> c**
(f x) y	**f :: a -> (b -> c)**

f x returns a function of type **b -> c**

g y	**g :: b -> c**

https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

ST Monad

```
type ST a = State -> (a, State)                    generalized ST

data ST0 a = S0 (State -> (a, State))

apply0  :: ST0 a -> State -> (a, State)
apply0 (S0 f) x = f x
apply0 (S0 g) x = g x

instance Monad ST0 where
  -- return :: a -> ST a
 return x   = S0 (\s -> (x,s))

  -- (>>=)  :: ST a -> (a -> STT b) -> ST b
  st >>= f   = S0 (\s -> let (x, s') = apply0 st s in apply0 (f x) s')
```

```
instance Monad ST where
   -- return :: a -> ST a
   return x  =  \s -> (x,s)

   -- (>>=)  :: ST a -> (a -> ST b) -> ST b
   st >>= f  =  \s -> let (x,s') = st s in f x s'
```

https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

Data Constructor

```
data Colour = Red | Green | Blue

data Colour = RGB Int Int Int

RGB :: Int -> Int -> Int -> Colour
```

https://stackoverflow.com/questions/18204308/haskell-type-vs-data-constructor

Examples (1)

```
pairs :: [a] -> [b] -> [(a,b)]                    do
pairs xs ys =  do x <- xs
                  y <- ys
                  return (x, y)
```

this function returns all possible ways of pairing elements from two lists

each possible value x from the list xs, and
each value y from the list ys, and
return the pair (x,y).

https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

Examples (1)

```
pairs :: [a] -> [b] -> [(a,b)]
pairs xs ys =  do x <- xs
                  y <- ys
                  return (x, y)
```

do

```
pairs xs ys = [(x,y) | x <- xs, y <- ys]
```

comprehension

In fact, there is a formal connection
between the do notation and the comprehension notation.
Both are simply different shorthands
for repeated use of the >>= operator for lists.

https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

Simple Examples (1)

```
(>>) :: Monad m => m a -> m b -> m b;
```

a1 >> a2 takes the actions a1 and a2 and
returns the mega action which is
a1-then-a2-returning-the-value-returned-by-a2.

https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

Simple Examples (1)

```
type State = Int

fresh :: ST0 Int
fresh =  S0 (\n -> (n, n+1))

wtf1 = fresh >>
          fresh >>
          fresh >>
          fresh

ghci> apply0 wtf1 0
```

https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

Simple Examples (2)

```
return :: a -> ST0 a

> wtf2 = fresh >>= \n1 ->
>        fresh >>= \n2 ->
>        fresh >>
>        fresh >>
>        return [n1, n2]

> wtf2' = do { n1 <- fresh;
>              n2 <- fresh;
>              fresh ;
>              fresh ;
>              return [n1, n2];
>            }
```

https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

Simple Examples (3)

```
ghci> apply0 wtf2 0



> wtf3 = do n1 <- fresh
>           fresh
>           fresh
>           fresh
>           return n1
```

https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

Dice Examples

to generate Int dice - result : a number between 1 and 6
throw results from a pseudo-random generator of type StdGen.

the type of the **state processors** will be

State StdGen Int

StdGen -> (Int, StdGen)

https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

randomR

the StdGen type : an instance of **RandomGen**

randomR :: (**Random** a, **RandomGen** g) => (a, a) -> g -> (a, g)

assume a is Int and g is StdGen

the type of **randomR**

randomR (1, 6) :: StdGen -> (Int, StdGen)

already have a **state processing function**

https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

randomR

```
randomR (1, 6) :: StdGen -> (Int, StdGen)

rollDie :: State StdGen Int
rollDie = state $ randomR (1, 6)
```

https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

Some Examples (1)

```
module StateGame where

import Control.Monad.State

-- Example use of State monad
-- Passes a string of dictionary {a,b,c}
-- Game is to produce a number from the string.
-- By default the game is off, a C toggles the
-- game on and off. A 'a' gives +1 and a b gives -1.
-- E.g
-- 'ab'    = 0
-- 'ca'    = 1
-- 'cabca' = 0
-- State = game is on or off & current score
--       = (Bool, Int)
```

https://wiki.haskell.org/State_Monad

Some Examples (2)

```
type GameValue = Int
type GameState = (Bool, Int)

playGame :: String -> State GameState GameValue
playGame []     = do
    (_, score) <- get
    return score
```

https://wiki.haskell.org/State_Monad

Some Examples (3)

```
playGame (x:xs) = do
    (on, score) <- get
    case x of
         'a' | on -> put (on, score + 1)
         'b' | on -> put (on, score - 1)
         'c'      -> put (not on, score)
         _        -> put (on, score)
    playGame xs

startState = (False, 0)

main = print $ evalState (playGame "abcaaacbbcabbab") startState
```

https://wiki.haskell.org/State_Monad

References

[1] ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf
[2] https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf