Monad P2 : State Transformer Generic Monad (1B)

Young Won Lim
10/2/19

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

A State Transformer

A State Transformer ST Example

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

a generic version of the **State monad** in **Control.Monad.State.Lazy**

a good example to learn **State** monad and general monads

do not be confused with **monad transformers**, **StateT**

and **Control.Monad.ST** (with reference variable **STRef**)

The **ST** monad in this example is similar to **StateT** monad

but is very different from the **ST** monad in **Control.Monad.ST**

State in Haskell, J. Launchbury, S. Pe Jones, 2016

https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/state-lasc.pdf

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)
```

generalized state transformers

return a result value in addition to the modified state

specify the result type as a parameter of the ST type

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

A Generic State Transformer (1)

the **state** may have **multiple components**

ex: multiple variables whose values we might want to update

→ use a different type for **State**

if we want two integers, we might use the type definition

```
type State = (Int, Int)
```

the standard library includes a module **Control.Monad.State**

that defines a parameterized version of

the **state-transformer monad**

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

A Generic State Transformer (2)

the functions which clients are allowed to use

```
module MyState (ST, get, put, apply) where
```

The type definition for a **generic state transformer** is very simple:

```
data ST s a = S (s -> (a, s))
```

a parameterized **state-transformer monad**
where the **state** is denoted by **type s** and
the **return value** of the transformer is the **type a**

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

A Generic State Transformer

```
type State = ...

type ST = State -> State

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

type ST State a =       State -> (a, State)
data ST State a = S ( State -> (a, State) )
```

generic state transformers

return a result value in addition to the modified state

specify the state & result type as a parameter of the **ST** type

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

A Generic State Transformer (3)

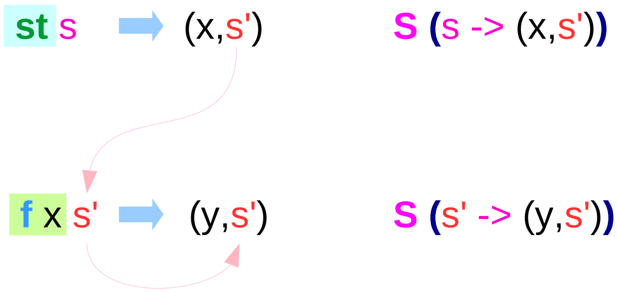
We make the above a monad by declaring it to be an **instance** of the **monad typeclass**

```
instance Monad (ST s) where
  return x   = S (\s -> (x, s))
  st >>= f   = S (\s -> let (x, s') = apply st s
                         in apply (f x) s')
```

where the function **apply** is just

```
apply :: ST s a -> s -> (a, s)
apply (S f) x = f x
```

```
st :: ST s a           :: S (\s -> (x, s))
f :: a -> ST s a       :: a -> S (\s -> (x, s))
```



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

A Generic State Transformer (4)

```
instance Monad (ST s) where
  return x   = S (\s -> (x, s))
  st >>= f   = S (\s -> let (x, s') = apply st s
                        in apply (f x) s')
```

the function **apply** is just
extracting the underlying state transformer function

```
apply :: ST s a -> s -> (a, s)
apply (S f) x = f x       pattern matching       apply (S f) = f
```

```
st :: ST s a              :: S (\s -> (x, s))
f :: a -> ST s a          :: a -> S (\s -> (x, s))
```

```
apply st s :: (a, s)
(x, s') :: (a, s)
```

```
f x :: ST s a             :: S (\s -> (x, s))
apply (f x) :: \s -> (a, s)
apply (f x) s' :: (a, s)
```

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

Accessing and modifying state (1)

a **get** and **put** function can access and modify the **state**.

Getting the current state via get

- an action that leaves the **state** unchanged,
- but returns the **state** itself as a **value**.

```
get = S (\s -> (s, s))
```

modifying the **state** to some new value **s'**

```
put s' = S (\_ -> ((), s'))
```

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

Accessing and modifying state (2)

```
fresh = S0 ( -> (n, n+1))

realfresh :: ST Int Int
realfresh = do n <- get
               put (n+1)
               return n
```

which denotes an action that ignores (ie blows away the old state) and replaces it with s'. Note that the put s' is an action that itselds yields nothing (that is, merely the unit value.)

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

Accessing and modifying state (2)

```
data ST s a = S (s -> (a, s))


realfresh :: ST Int Int                 :: S (Int -> (Int, Int))
realfresh = do n <- get
               put (n+1)
               return n
```

```
get         = S (\s -> (s, s))
put s'      = S (\_ -> ((), s'))
return x    = S (\s -> (x, s))
```

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

Using a Generic State Transformer (1)

using generic state monad to the tree labeling function

Note that the actual type definition of the generic transformer

is hidden from us, so we must use only the publicly exported functions:

get, **put** and **apply** (in addition to the default monadic functions)

the action that returns the next fresh integer.

Using the generic state-transformer, we write it as:

```
freshS :: ST Int Int
freshS = do n <- get
            put (n+1)
            return n
```

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

Using a Generic State Transformer (2)

Now, the labeling function is straightforward

```
mlabelS :: Tree a -> ST Int (Tree (a,Int))
mlabelS (Leaf x)   =  do n <- freshS
                         return (Leaf (x, n))
mlabelS (Node l r) =  do l' <- mlabelS l
                         r' <- mlabelS r
                         return (Node l' r')
```

```
ghci> apply (mlabelS tree) 0
(Node (Node (Leaf ('a', 0)) (Leaf ('b', 1))) (Leaf ('c', 2)), 3)
```

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

Using a Generic State Transformer (2)

We can execute the action from any initial state of our choice

```
ghci> apply (mlabelS tree) 1000
(Node (Node (Leaf ('a',1000)) (Leaf ('b',1001))) (Leaf ('c',1002)),1003)
```

Now, whats the point of a generic state transformer
if we can’t have richer states.
Next, let us extend our fresh and label functions so that

> each node gets a new label (as before),
> the state also contains a map of the frequency
> with which each leaf value appears in the tree.

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

Using a Generic State Transformer (3)

Thus, our state will now have two elements,
an integer denoting the next fresh integer,
and a Map a Int denoting the number of times
each leaf value appears in the tree.

```
data MySt a = M { index :: Int
                , freq  :: Map a Int }
              deriving (Eq, Show)
```

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

Using a Generic State Transformer (4)

We write an action that returns the next fresh integer as

```
freshM = do
  s     <- get
  let n  = index s
  put $ s { index = n + 1 }
  return n
```

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

Using a Generic State Transformer (5)

we want an action that updates the frequency of a given element k

```
updFreqM k = do
  s    <- get
  let f = freq s
  let n = findWithDefault 0 k f
  put $ s {freq = insert k (n + 1) f}
```

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

Using a Generic State Transformer (6)

And with these two, we are done

```
mlabelM (Leaf x)   =  do updFreqM x
                         n <- freshM
                         return $ Leaf (x, n)

mlabelM (Node l r) =  do l' <- mlabelM l
                         r' <- mlabelM r
                         return $ Node l' r'
```

Now, our initial state will be something like

```
initM = M 0 empty
```

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

Using a Generic State Transformer (6)

and so we can label the tree

```
ghci> let tree2   = Node tree tree
ghci> let (lt, s) = apply (mlabelM tree) $ M 0 empty

ghci> lt
Node (Node (Node (Leaf ('a',0)) (Leaf ('b',1))) (Leaf ('c',2)))
(Node (Node (Leaf ('a',3)) (Leaf ('b',4))) (Leaf ('c',5)))

ghci> s
M {index = 6, freq = fromList [('a',2),('b',2),('c',2)]}
```

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

References

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

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