State Monad

From HaskellWiki

The State Monad by Example

```
import Control.Monad.State.Lazy (https://hackage.haskell.org/package/mtl/docs/Control-Monad-State-Lazy.html)
```

This is a short tutorial on the state monad. Emphasis is placed on intuition. The types have been simplified to protect the innocent.

Another longer walkthrough of the state monad can be found in the wiki book section Understanding monads/State. (https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State)

Contents

- 1 Foundations
  - 1.1 Primitives
  - 1.2 Combinations
  - 1.3 Other Functions
- 2 Implementation
- 3 Complete and Concrete Example 1
- 4 Complete and Concrete Example 2

Foundations

Primitives

```
runState (return 'X') 1
```

('X',1)

`return` set the result value but leave the state unchanged. Comments:

```
  return 'X' :: State Int Char
  runState (return 'X') :: Int -> (Char, Int)
  initial state = 1 :: Int
  final value = 'X' :: Char
  final state = 1 :: Int
  result = ('X', 1) :: (Char, Int)
```

```
runState get 1
```

(1,1)

`get` set the result value to the state and leave the state unchanged. Comments:

```
get :: State Int Int
runState get :: Int -> (Int, Int)
initial state = 1 :: Int
final value = 1 :: Int
final state = 1 :: Int
```

```
runState (put 5) 1
```

((),5)

`put` set the result value to () and set the state value. Comments:

```
put 5 :: State Int ()
runState (put 5) :: Int -> ((),Int)
initial state = 1 :: Int
final value = () :: ()
final state = 5 :: Int
```

Combinations

Because (State s) forms a monad, values can be combined with (>>=) or do{}.

```
runState (do { put 5; return 'X' }) 1
```

('X',5)

Comments:

```
do { put 5; return 'X' } :: State Int Char
runState (do { put 5; return 'X' }) :: Int -> (Char,Int)
initial state = 1 :: Int
final value = 'X' :: Char
final state = 5 :: Int
```

```
postincrement = do { x <- get; put (x+1); return x }
runState postincrement 1
```

(1,2)

```
predecrement = do { x <- get; put (x-1); get }
runState predecrement 1
```

(0,0)

Other Functions

```
runState (modify (+1)) 1
```

((),2)

```
runState (gets (+1)) 1
```

(2,1)

```
evalState (gets (+1)) 1
```

2

```
execState (gets (+1)) 1
```

1

Implementation

At its heart, a value of type (State s a) is a function from initial state s to final value a and final state s: (a,s). These are usually wrapped, but shown here unwrapped for simplicity.

Return leaves the state unchanged and sets the result:

```
-- ie: (return 5) 1 -> (5,1)
return :: a -> State s a
return x s = (x,s)
```

Get leaves state unchanged and sets the result to the state:

```
-- ie:  get 1 -> (1,1)
get :: State s s
get s = (s,s)
```

Put sets the result to () and sets the state:

```
-- ie: (put 5) 1 -> ((),5)
put :: s -> State s ()
put x s = ((),x)
```

The helpers are simple variations of these primitives:

```
modify :: (s -> s) -> State s ()
modify f = do { x <- get; put (f x) }

gets :: (s -> a) -> State s a
gets f = do { x <- get; return (f x) }
```

EvalState and execState just select one of the two values returned by runState. EvalState returns the final result while execState returns the final state:

```
evalState :: State s a -> s -> a
evalState act = fst . runState act

execState :: State s a -> s -> s
execState act = snd . runState act
```

Combining two states is the trickiest bit in the whole scheme. To combine do { x <- act1; fact2 x } we need a function which takes an initial state, runs act1 to get an intermediate result and state, feeds the intermediate result to fact2 and then runs that action with the intermediate state to get a final result and a final state:

```
(>>=) :: State s a -> (a -> State s b) -> State s b
(act1 >>= fact2) s = runState act2 is
    where (iv,is) = runState act1 s
          act2 = fact2 iv
```

Complete and Concrete Example 1

Simple example that demonstrates the use of the standard Control.Monad.State monad. It's a simple string parsing algorithm.

```
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)

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

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

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
```

Complete and Concrete Example 2

```haskell
-- a concrete and simple example of using the State monad


import Control.Monad.State

-- non monadic version of a very simple state example
-- the State is an integer.
-- the value will always be the negative of the state

type MyState = Int

valFromState :: MyState -> Int
valFromState s = -s
nextState :: MyState->MyState
nextState x = 1+x

type MyStateMonad  = State MyState

-- this is it, the State transformation.  Add 1 to the state, return -1*the state as the computed value.
getNext ::   MyStateMonad  Int
getNext  = state  (\st -> let st' = nextState(st) in (valFromState(st'),st') )



-- advance the state three times.
inc3::MyStateMonad Int
inc3 =  getNext >>= \x ->
        getNext >>= \y ->
        getNext >>= \z ->
        return z

-- advance the state three times with do sugar
inc3Sugared::MyStateMonad Int
inc3Sugared = do x <- getNext
                 y <- getNext
                 z <- getNext
                 return z

-- advance the state three times without inspecting computed values
inc3DiscardedValues::MyStateMonad Int
inc3DiscardedValues =    getNext >> getNext >> getNext

-- advance the state three times without inspecting computed values with do sugar
inc3DiscardedValuesSugared::MyStateMonad Int
inc3DiscardedValuesSugared =    do
                            getNext
                            getNext
                            getNext


-- advance state 3 times, compute the square of the state
inc3AlternateResult::MyStateMonad Int
inc3AlternateResult = do  getNext
                          getNext
                          getNext
                          s<-get
                          return (s*s)


-- advance state 3 times, ignoring computed value, and then once more
inc4::MyStateMonad Int
inc4 = do
          inc3AlternateResult
          getNext

main =
     do
          print (evalState inc3 0)                          -- -3
          print (evalState inc3Sugared 0)                   -- -3
          print (evalState inc3DiscardedValues 0)           -- -3
          print (evalState inc3DiscardedValuesSugared 0)    -- -3
          print (evalState inc3AlternateResult 0)           -- 9
          print (evalState inc4 0)                          -- -4
```

Retrieved from "https://wiki.haskell.org/index.php?title=State_Monad&oldid=62675"