Monad P2: State Monad Basics (2A)

Young Won Lim
7/6/19

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

Type Synonyms

```
type String = [Char]

phoneBook :: [(String,String)]
```

```
type PhoneBook = [(String,String)]

phoneBook :: PhoneBook
```

```
type PhoneNumber = String
type Name = String
type PhoneBook = [(Name,PhoneNumber)]

phoneBook :: PhoneBook
```

```
phoneBook =
    [("betty","555-2938")
    ,("bonnie","452-2928")
    ,("patsy","493-2928")
    ,("lucille","205-2928")
    ,("wendy","939-8282")
    ,("penny","853-2492")
    ]
```

http://learnyouahaskell.com/making-our-own-types-and-typeclasses

Record Syntax (named field)

```
data Configuration = Configuration { username :: String  }

let cfg = Configuration  { username = "ABCD"   }

username  cfg  ➔ “ABCD”
```

```
Configuration :: String -> Configuration

username :: Configuration -> String
```

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

let stst = State { runState = (\y -> (y, y+1))  }

runState stst  ➔  (\y -> (y, y+1))
```

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

runState :: State s a -> (s -> (s, a))
```

https://en.wikibooks.org/wiki/Haskell/More_on_datatypes

Record Syntax

```
data Configuration = Configuration { username :: String  }
```

```
Configuration "ABCD"   ➡   cfg :: Configuration
username  cfg          ➡   "ABCD" :: String
```

```
Configuration { username = "ABCD"  }
```

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

```
State (\y -> (y, y+1))   ➡   stst  :: State s a
runState stst            ➡    (\y -> (y, y+1)) :: s -> (s, a)
```

```
State { runState = (\y -> (y, y+1))  }
```

https://en.wikibooks.org/wiki/Haskell/More_on_datatypes

Record Syntax – type signatures

```
data Configuration = Configuration { username :: String }
```

```
data Configuration = Configuration { username :: String }
```

```
Configuration :: String -> Configuration
```

```
username :: Configuration -> String
```

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

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

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

```
runState :: State s a -> (s -> (s, a))
```

https://en.wikibooks.org/wiki/Haskell/More_on_datatypes

A Wrapper Type

State Monad :

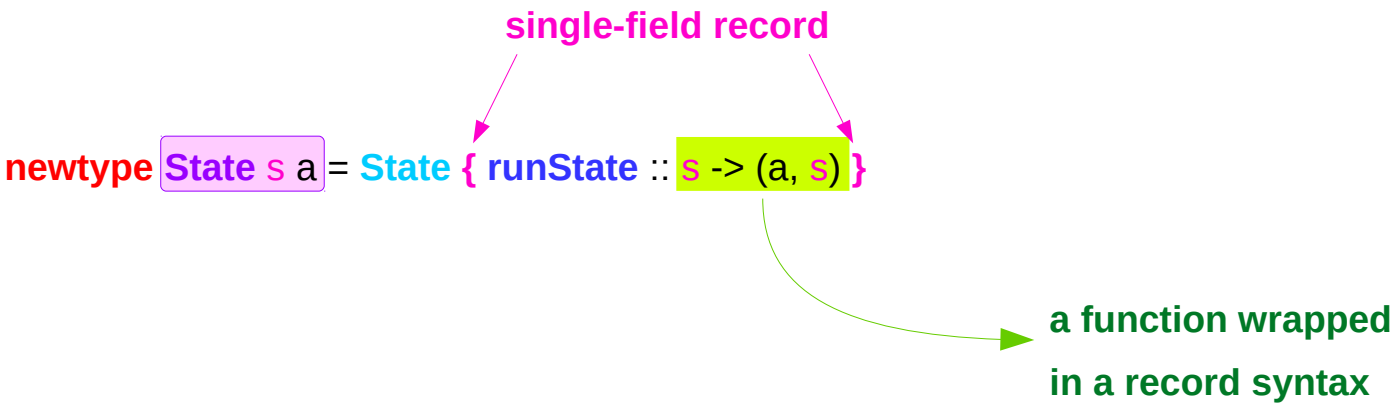
- a simple wrapper type
- usually defined with **newtype**.

type : type synonyms for an existing type (no data constructor)

newtype : can make an instance

A single **data constructor** : **State { runState :: s -> (s, a) }**

A single **field** : **{ runState :: s -> (s, a) }**



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

newtype and **data**

data  **newtype**

data can only be replaced with newtype
if the type has exactly *one constructor*
with exactly *one field* inside it.

a single constructor and a single field
allow the **compiler to remove**
the trivial **wrapping** and **unwrapping**
operations for the single field
(**no runtime overhead**)

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

data, **type**, and **newtype**

```
data     State s a = State { runState :: s -> (s, a) }
type     State s a = State { runState :: s -> (s, a) }
newtype  State s a = State { runState :: s -> (s, a) }
```

data	**instance**	**overhead**
type	**N/A**	**N/A**
newtype	**instance**	**N/A**

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

newtype examples

```
newtype Fd = Fd Cint          (O)
newtype Fd = Fd Cint          (O)

newtype Identity a = Identity a  deriving (Eq, Ord, Read, Show)    (O)

newtype State s a = State { runState :: s -> (s, a) }              (O)

newtype Pair a b = Pair { pairFst :: a, pairSnd :: b }             (X)
data    Pair a b = Pair { pairFst :: a, pairSnd :: b }             (O)
newtype NPair a b = NPair (a, b)                                   (O)
```

newtype enables an instance,
deriving clauses

Newtype enables the record
with only one constructor
and one field

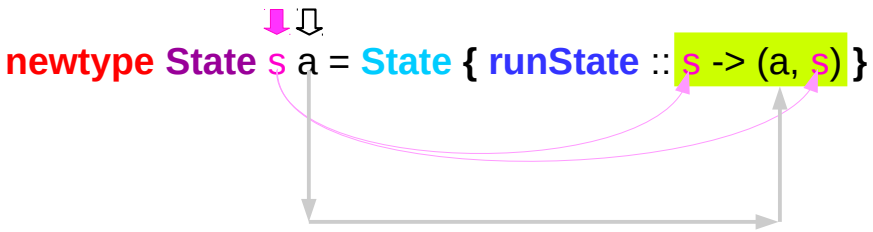
2 fields not allowed in **newtype**

2 fields allowed in **data**

1 field : ordered pair

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

Parameterized type **State**



s : the type of the state,

a : the type of the produced result

s -> (a, s) : function type

State String,

State Int,

State SomeLargeDataStructure,

and so forth.

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

The wrapped function

Calling the type **State** looks like a misnomer

because the **wrapped value** is not the state itself

but a **state processor** (**accessor function**: **runState**)

(a **function** is treated as a **value** in Haskell)

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

The **function** is also a **value**

The **wrapped value** is a **function**

(**state processor** of the type **s -> (s, a)**)

state processor :: (s -> (s, a))

data constructor

State :: (s -> (s, a)) -> State s a

data constructor

accessor function

runState :: State s a -> (s -> (s, a))

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

The state function in a record

The Haskell **type State** describes a **state processor** :: **s -> (s, a)**

that take a state — s

and return both a result and an updated state, — a, s

which are given back in a tuple. — (a, s)

The **state function** is wrapped
by a **data** type definition (usually **newtype**)
with a **runState accessor**

state processor :: :: **s -> (s, a)**

wrapped in
a **record** with
a single **field**

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

accessor function

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

Making a value – using the data constructor State

State : **data constructor** for single **record** with single **field**

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

```
let stst = State { runState = (\y -> (y, y+1))  }
```

a record syntax

- must make an **accessor** function using **pattern matching**

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

Making a value – using the function "state"

in practices, State **data constructor** is not allowed to be accessed

Instead, the function **state** is provided

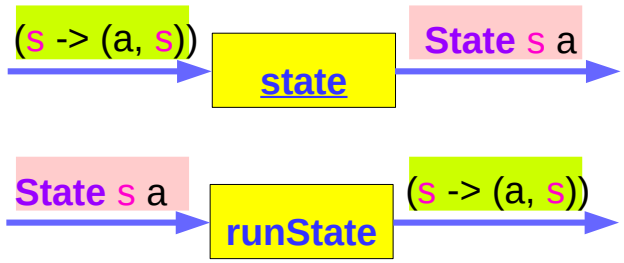
```
newtype State s a = State { runState :: s -> (a, s) }
```

```
let stst = state (\y   -> (y, y+1))
```

a library function

- the accessor function **runState** is provided

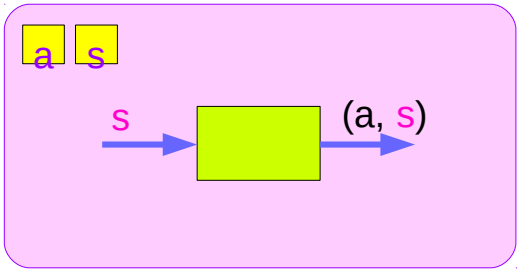
Control.Monad.Trans.State
exports a **state** function



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

Accessor Function **runState**

```
1) let stst = State { runState = (\y -> (y, y+1))  }

2) let stst = state (\y -> (y, y+1))

runState stst          ➡   (\y -> (y, y+1))  -- no instance error
runState stst  1       ➡   (1, 2)
```



stst :: **State** s a

a s binding variable type

state processor

run State Processor (Function)

State Packages

Control.Monad.**Trans**.**State**,	**transformers** package. (focused here)
Control.Monad.**State**,	**mtl** (Monad Transformer Library) package.
Control.Monad.**State.Lazy**,	**mtl** (Monad Transformer Library) package.

```
import Control.Monad.Trans.State

import Control.Monad.State
import Control.Monad.State.Lazy
```

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

Transformer Packages

transformers: Concrete functor and monad transformers

This package contains:

- the monad transformer class (in Control.Monad.Trans.Class)
- concrete functor and monad transformers,
- each with associated operations and functions
  to lift operations associated with other transformers.

http://hackage.haskell.org/package/transformers

Transformer Packages

The package can be used on its own in portable Haskell code,
in which case operations need to be **manually lifted** through transformer stacks
(see Control.Monad.Trans.Class for some examples).

Alternatively, it can be used with the non-portable monad classes
in the **mtl** or **monads-tf** packages, which automatically lift operations
introduced by monad transformers through other transformers.

http://hackage.haskell.org/package/transformers

Monad Transformer Class

A monad transformer makes a new monad out of an existing monad,
such that computations of the old monad may be embedded in the new one.

To construct a monad with a desired set of features,
one typically starts with a base monad, such as Identity, [] or IO,
and applies a sequence of monad transformers.

http://hackage.haskell.org/package/transformers-0.5.6.2/docs/Control-Monad-Trans-Class.html

Monad Transformer Class

class MonadTrans t where

The class of monad transformers.
Instances should satisfy the following laws,
which state that lift is a monad transformation:

```
lift . return = return

lift (m >>= f) = lift m >>= (lift . f)
```

Methods

```
lift :: Monad m => m a -> t m a
```

Lift a computation from the argument monad to the constructed monad.

http://hackage.haskell.org/package/transformers-0.5.6.2/docs/Control-Monad-Trans-Class.html

The “**state**” function

Control.Monad.**Trans**.**State**

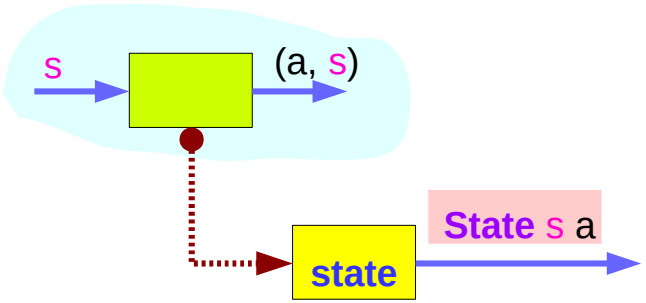
no State data constructor

instead the function “**state**”

```
state :: (s -> (a, s)) -> State s a
```

Control.Monad.**State**

different implements of the **State**



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

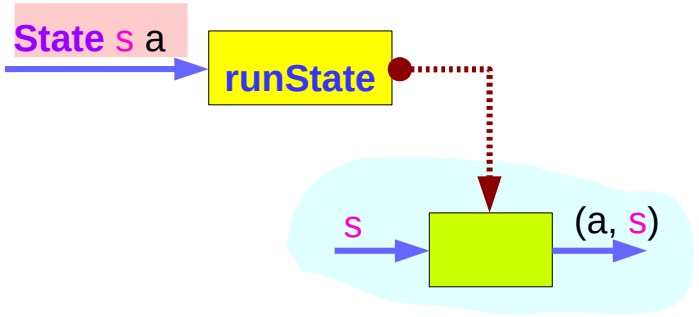
runState function

State is a record with only one element,
whose type is a function (:: s -> (a, s))

runState converts a value of type State s a
to a function of this type (:: s -> (a, s))

runState :: State s a -> s -> (a, s)

apply **runState** to a value of the type **State** s a,
the return type is a function type **s -> (a, s)**

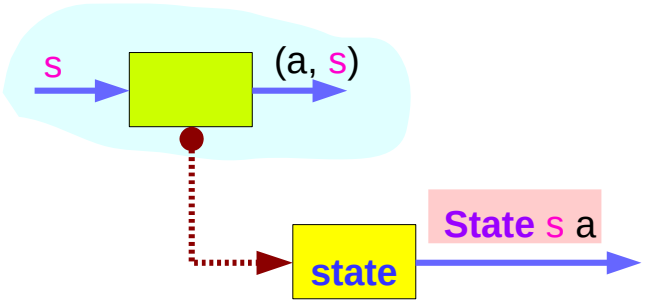



```
newtype State s a = State { runState :: s -> (a, s) }
```

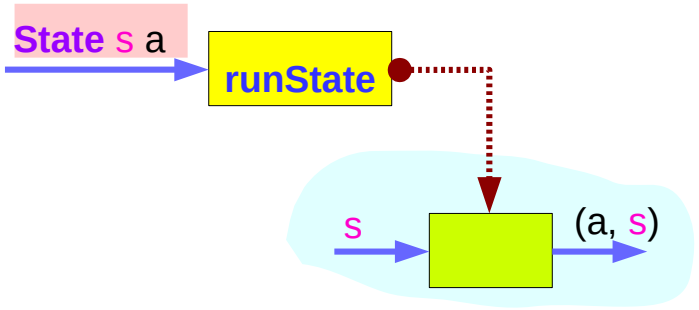
https://stackoverflow.com/questions/3240947/understanding-haskell-accessor-functions

state & runState functions

```
state :: s -> (a, s) -> State s a
```



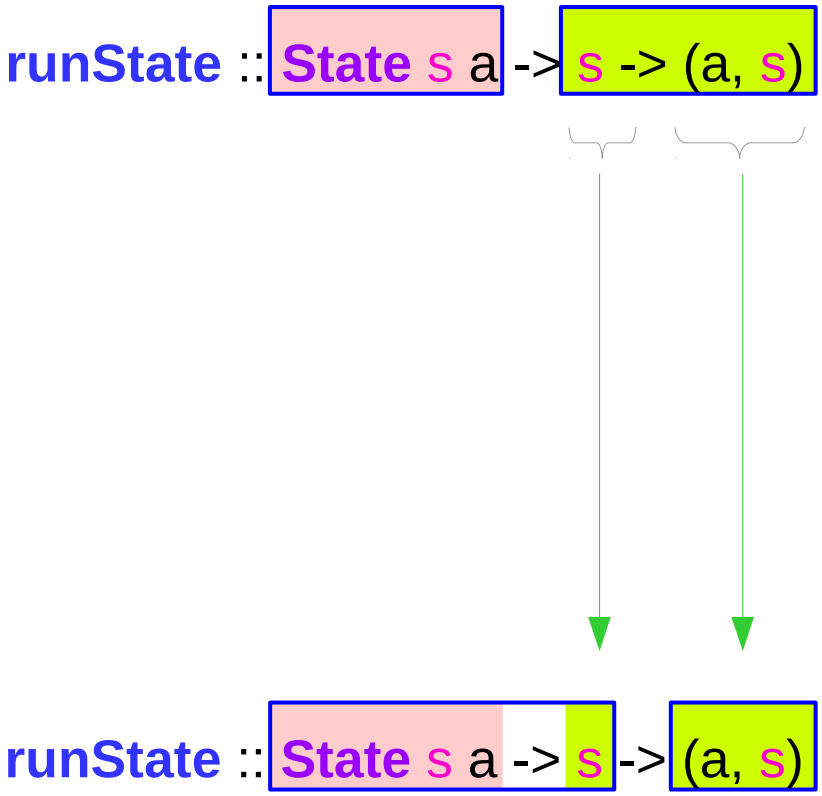

```
runState :: State s a -> s -> (a, s)
```

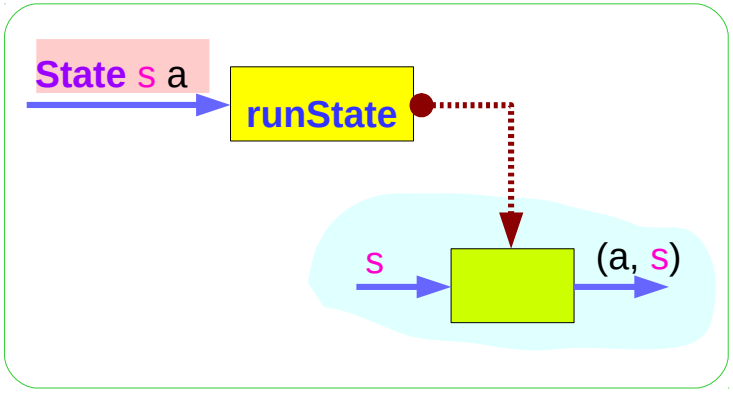


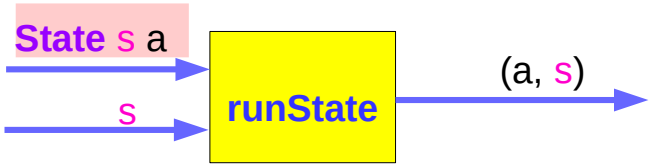

```
newtype State s a = State { runState :: s -> (a, s) }
```

https://stackoverflow.com/questions/3240947/understanding-haskell-accessor-functions

runState function – partially applied








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

https://stackoverflow.com/questions/3240947/understanding-haskell-accessor-functions

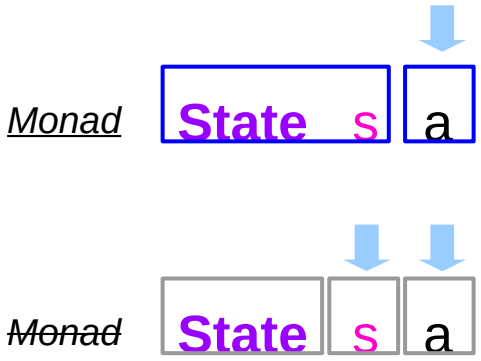
Instantiating a State Monad

wrap a function type and give it a **name**. s -> (a, s)

State s can be made into a *Monad instance*, for every type s

the type of a *Monad instance* is **State** s

State can not be made into a *Monad instance*,
(as that instance would take two type parameters, rather than one.)



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

Monad instance of State s

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

```
instance Monad (State s)  where
    return implementation
    (>>=) implementation
```

```
1) let stst = State { runState = (\y -> (y, y+1))  }     a way of thinking     a record construction
2) let stst = state (\y -> (y, y+1))                     an actual way         a library function
```

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

Common implementation of **return** and **>>=**

instance Monad (State s) where

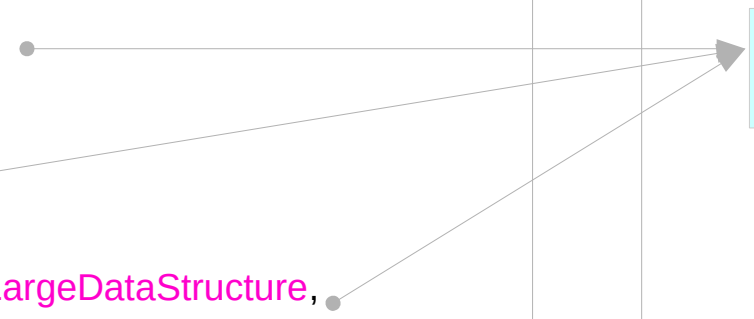
many different **State** monads,
one for each possible type of state -

State String,

State Int,

State SomeLargeDataStructure,

and so forth.



one implementation of

- **return**
- **(>>=)**

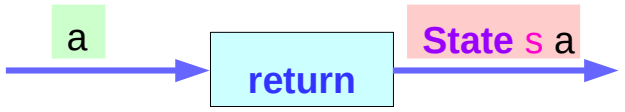
can handle these different (**State** s) monads
according to different choices of s.

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

return method

```
instance Monad (State s) where

return :: a -> State s a
return x = state ( \s -> (x, s) )
```

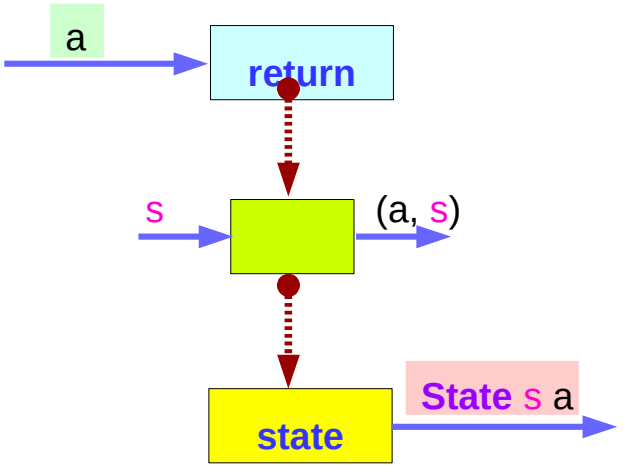
→ State s a



giving a value (x) to **return**
results in a **state processor** function

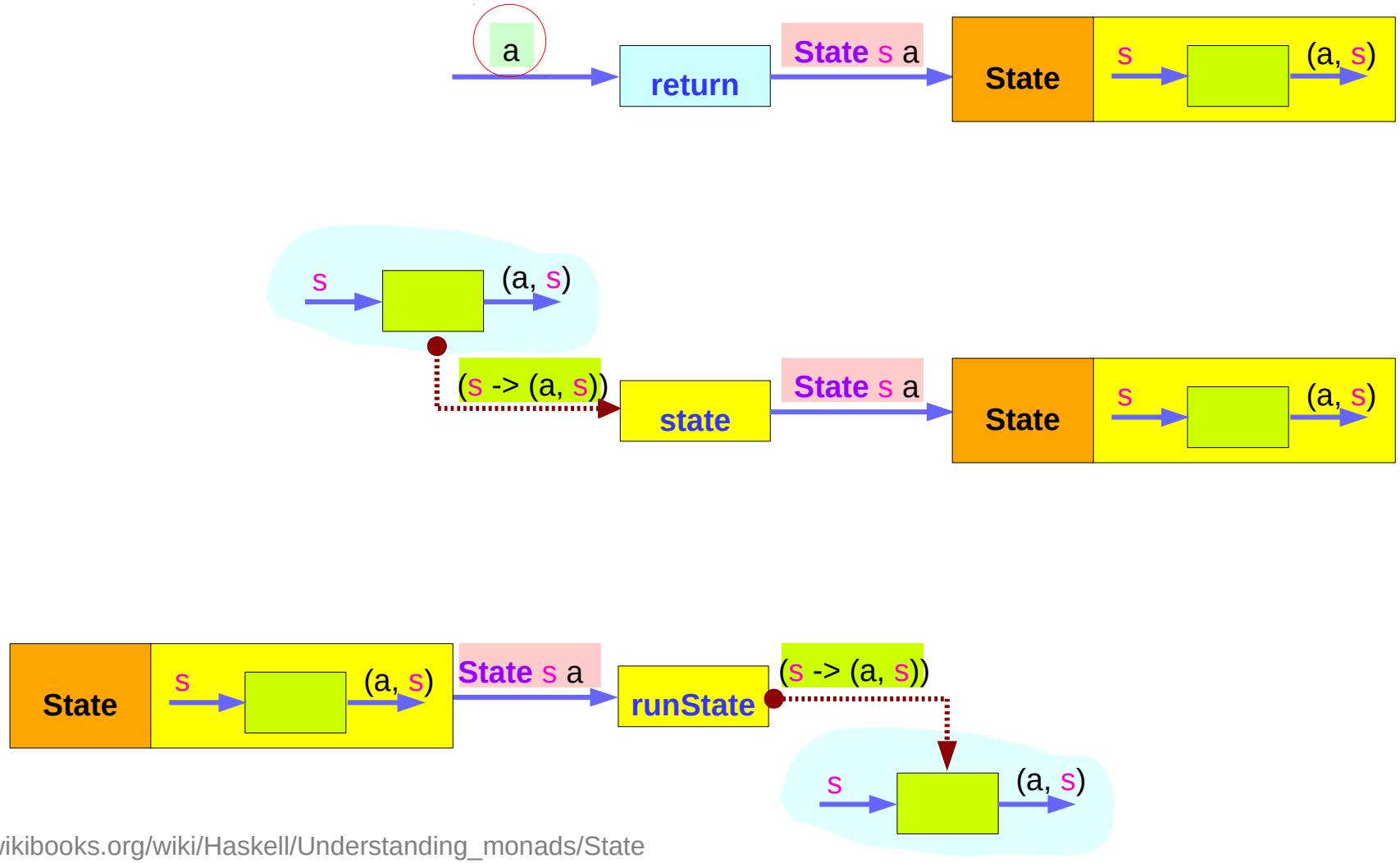
which takes a state (s) and
returns it unchanged (s),
together with the value x

finally, the function is wrapped up by **state.**



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

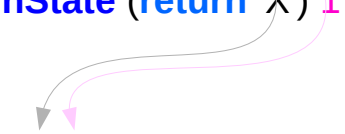
return, state, runState methods



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

State Monad Examples – return

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



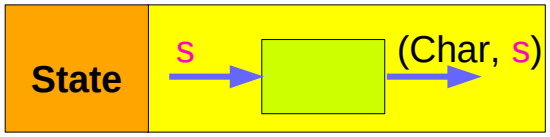
```
('X',1)
```

return sets the result *value* but leave the *state* unchanged.

```
return 'X' :: State Int Char
runState (return 'X') :: Int -> (Char, Int)
runState (return 'X') 1  ➡ ('X', 1)

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

https://wiki.haskell.org/State_Monad

```
return 'X' :: State Int Char
```




```
runState (return 'X')  :: Int -> (Char, Int)
```


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

runState, evalState and execState

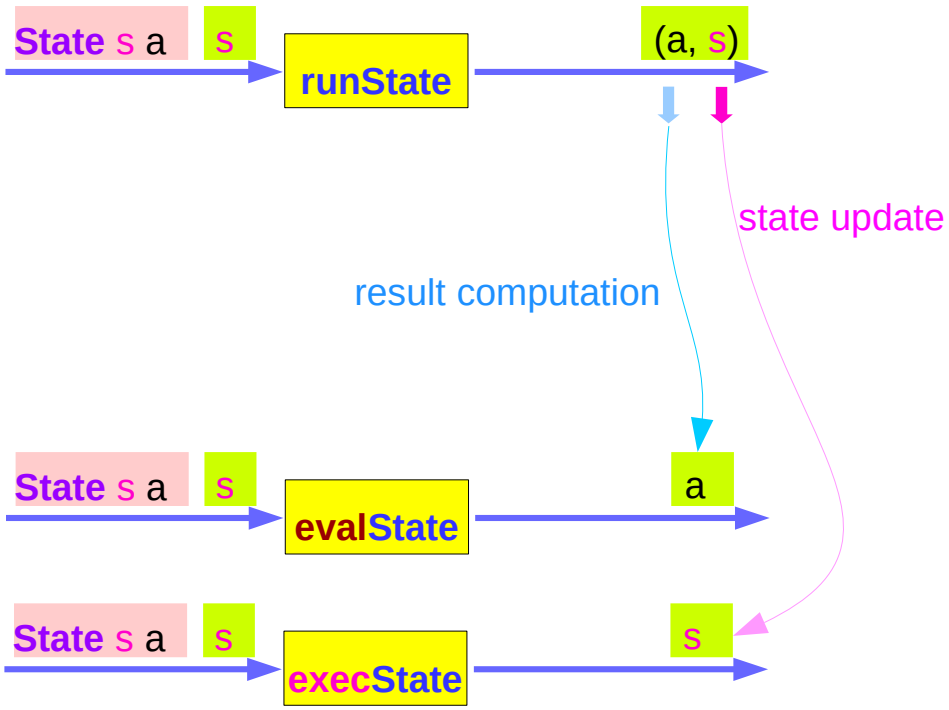
runState

unwrap the State s a value

to get the actual **state processing function**

then it is applied to some initial state

to get the tuple (result, updated state)



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

evalState and execState

Given a State s a and an initial state s,

evalState only the result value

execState just the new state.

p :: **State** s a

s :: s

p s ➡ (a, s)

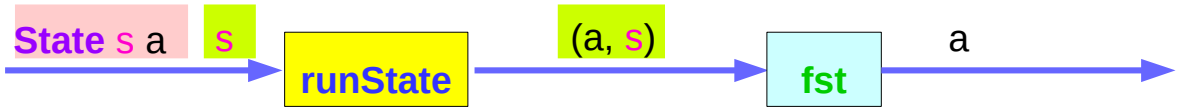
```
evalState :: State s a -> s -> a
evalState p s = fst (runState p s)
```

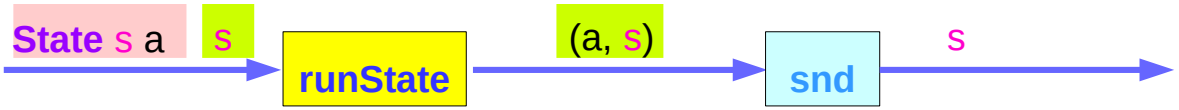

```
execState :: State s a  -> s -> s
execState p s = snd (runState p s)
```

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

fst and snd in evalState and execState

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

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





https://wiki.haskell.org/State_Monad

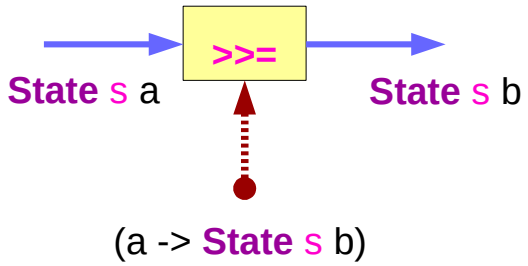
Function type of >>=

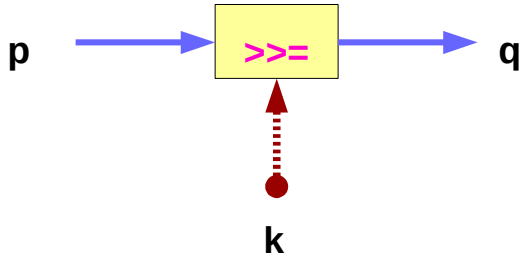
```
instance Monad (State s) where

(>>=) :: State s a -> (a -> State s b) -> State s b
p >>= k = q where
```

p :: **State** s a

k :: (a -> **State** s b)

State s a ->	(a -> **State** s b) ->	**State** s b
p	**k**	**p >>= k**

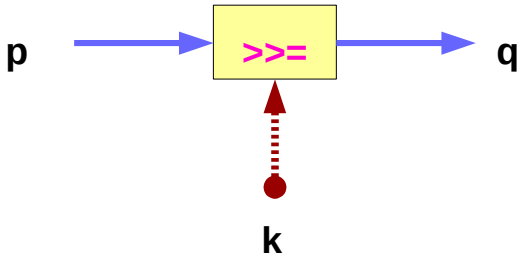




https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

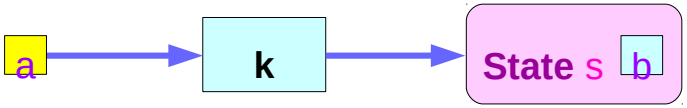
1st and 2nd arguments of >>= :

```
instance Monad (State s) where

(>>=) :: State s a -> (a -> State s b) -> State s b
p >>= k = q where
```



p :: **State** s a

k :: (a -> **State** s b)



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

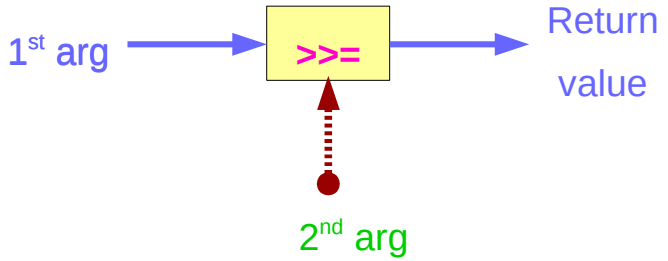
Binding operator >>=

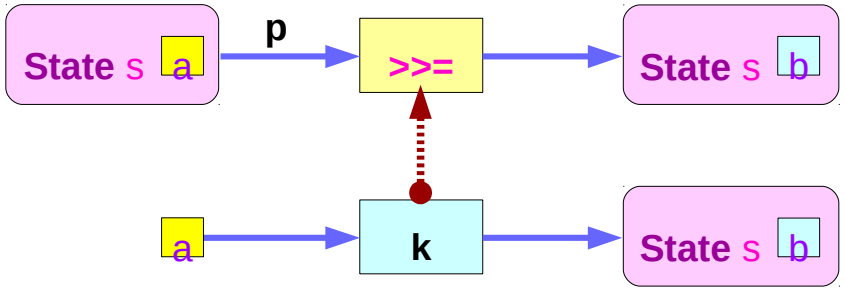
```
instance Monad (State s) where

(>>=) :: State s a -> (a -> State s b) -> State s b
p >>= k = q where
```

p :: **State** s a — State Monad value

k :: (a -> **State** s b) — State Monad returning function

p >>= k = q



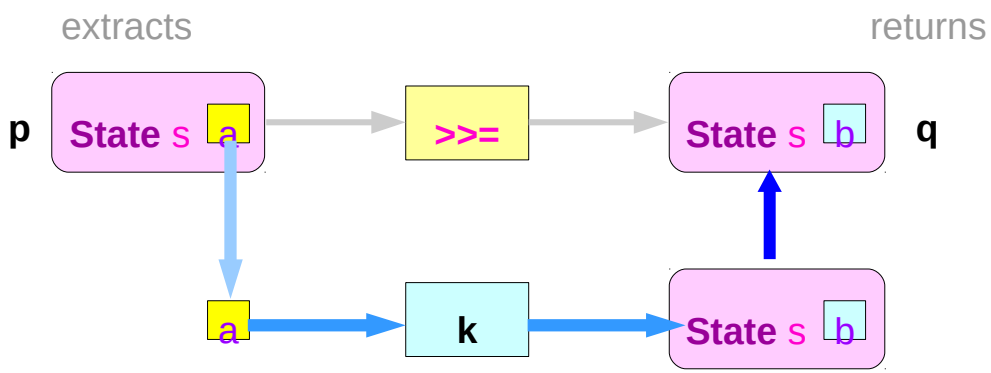


https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

Conceptual computation flow of >>=

```
instance Monad (State s) where

(>>=) :: State s a -> (a -> State s b) -> State s b
p >>= k = q where
```

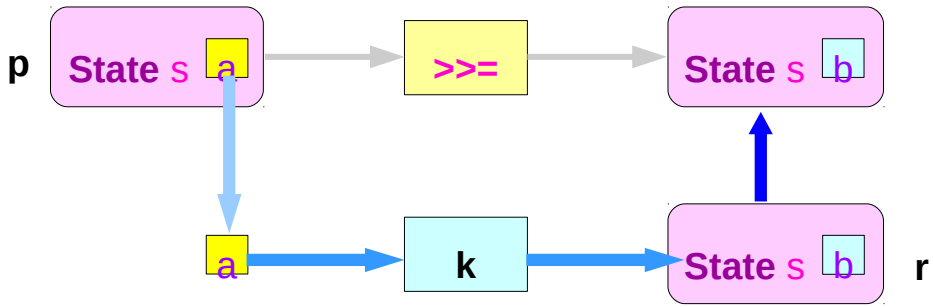
state transition : running the state processor



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

Unwrapping the state processing function : **runState**

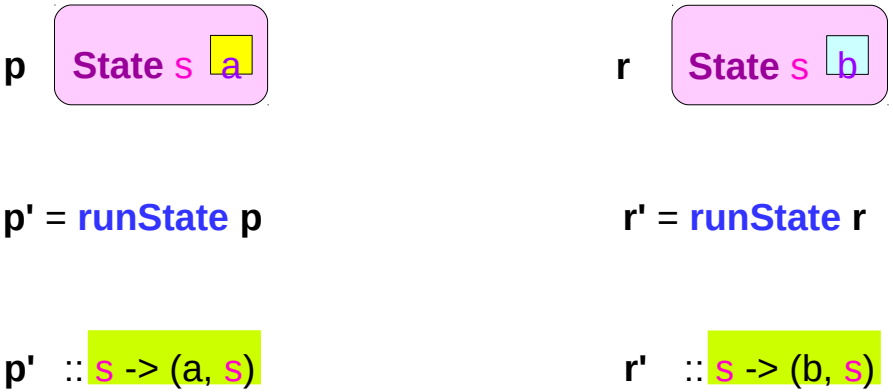
```
instance Monad (State s) where

(>>=) :: State s a -> (a -> State s b) -> State s b
p >>= k = q where
    p' = runState p          -- p' :: s -> (a, s)
    k' = runState . k        -- k' :: a -> s -> (b, s)
```



runState unwraps

state processors

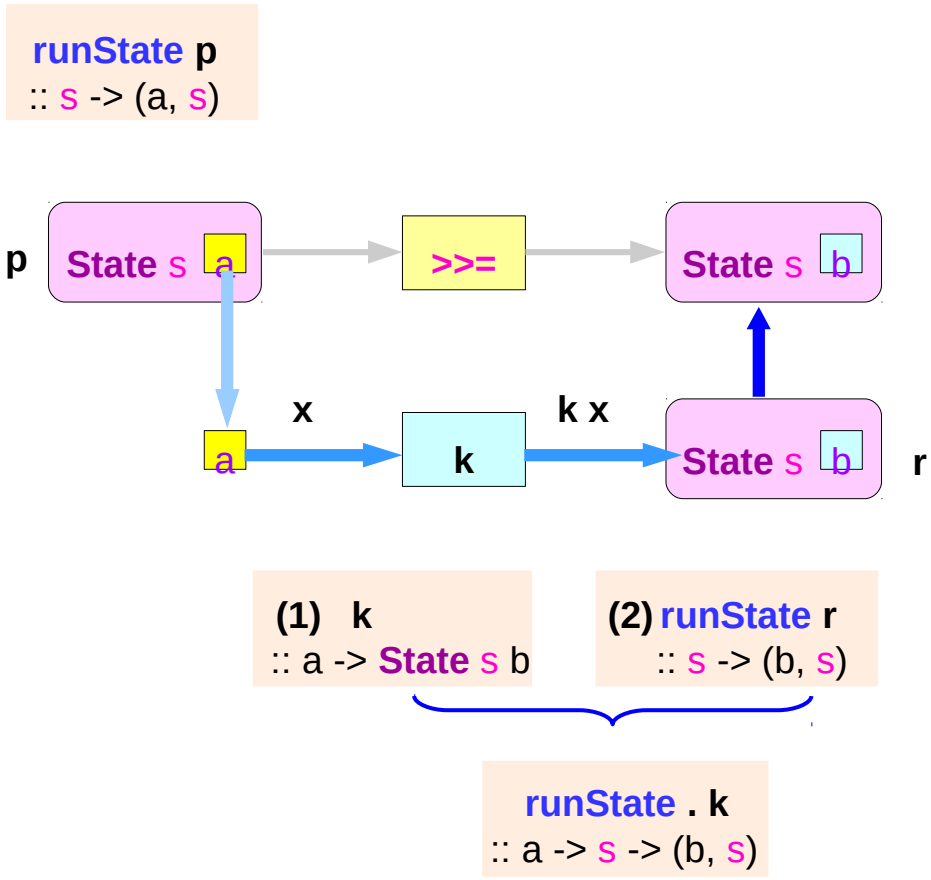
p' and **r'**



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

Composite Function runState . k

```
instance Monad (State s) where

(>>=) :: State s a -> (a -> State s b) -> State s b
p >>= k = q where
   p' = runState p          -- p' :: s -> (a, s)
   k' = runState . k        -- k' :: a -> s -> (b, s)
```




```
x :: a
k :: a -> State s b
k x :: State s b
r = k x :: State s b
runState r  ::  s -> (b, s)
runState p ::  s -> (a, s)
```

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

runState and runState . (k x)

```
instance Monad (State s) where

(>>=) :: State s a -> (a -> State s b) -> State s b
p >>= k = q where
    p' = runState p          -- p' :: s -> (a, s)
    k' = runState . k        -- k' :: a -> s -> (b, s)
```

runState p
:: s -> (a, s)

runState . k
:: a -> s -> (b, s)

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

State Transitions

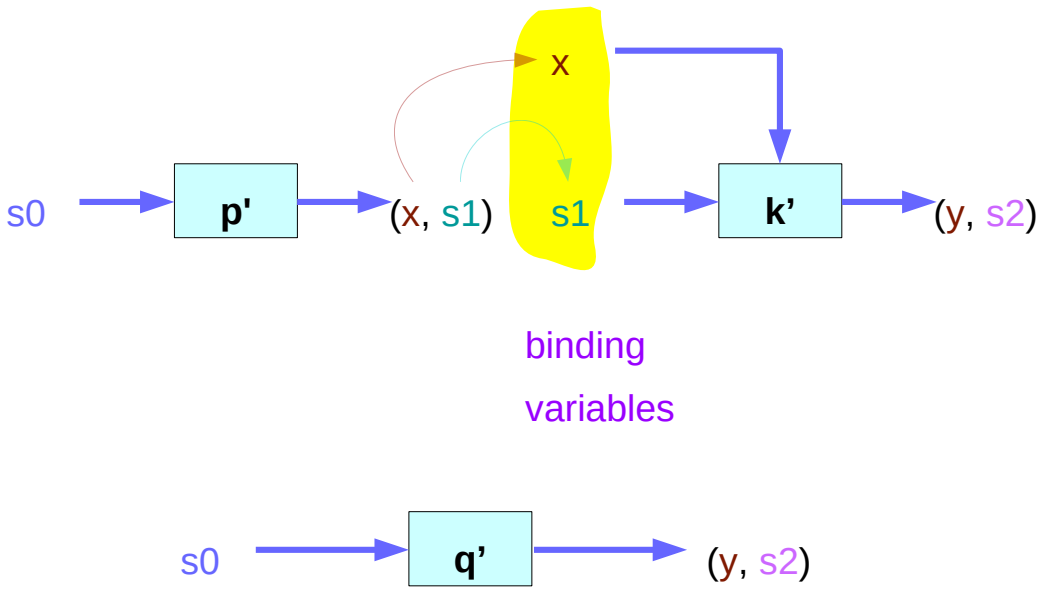
```
instance Monad (State s) where

(>>=) :: State s a -> (a -> State s b) -> State s b
p >>= k = q where
    p' = runState p          -- p' :: s -> (a, s)
    k' = runState . k        -- k' :: a -> s -> (b, s)
    q' s0 = (y, s2) where    -- q' :: s -> (b, s)
        (x, s1) = p' s0      -- (x, s1) :: (a, s)
        (y, s2) = k' x s1    -- (y, s2) :: (b, s)
    q = state q'
```

(?, s0) → (x, S1) → (y, s2)

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

State Transition from s0 to s2

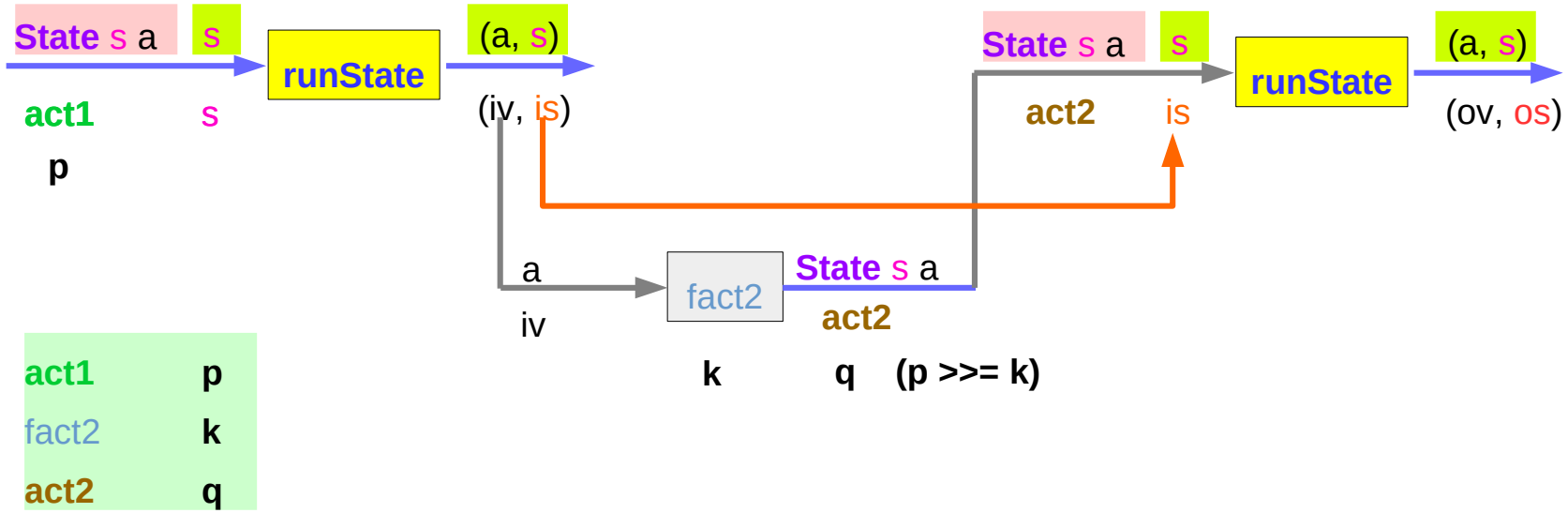
```
instance Monad (State s) where

(>>=) :: State s a -> (a -> State s b) -> State s b
p >>= k = q where
    p' = runState p              -- p' :: s -> (a, s)
    k' = runState . k            -- k' :: a -> s -> (b, s)
    q' s0 = (y, s2) where        -- q' :: s -> (b, s)
        (x, s1) = p' s0          -- (x, s1) :: (a, s)
        (y, s2) = k' x s1        -- (y, s2) :: (b, s)
    q = state q'
```



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

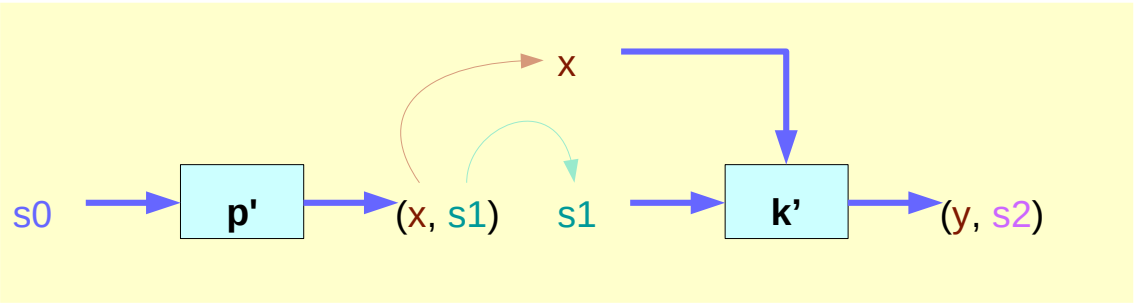
Unwrapped Implementation Examples

```
(>>=) :: 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
```

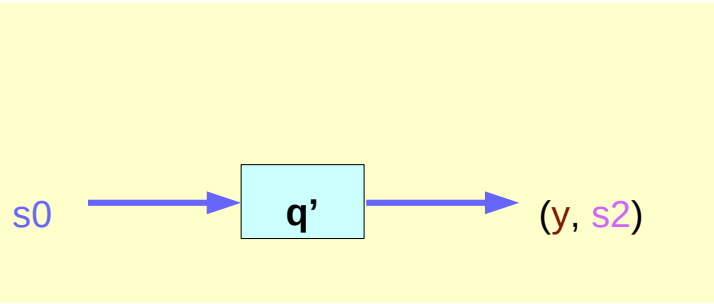


https://wiki.haskell.org/State_Monad

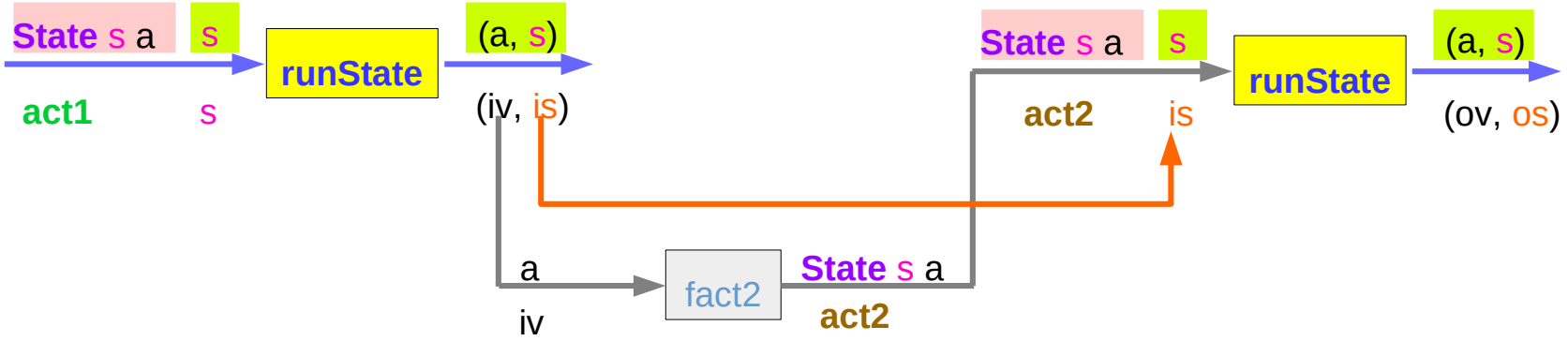
State Transition from s0 to s2



(*, s) (iv, is) (ov, os)



(*, s) (ov, os)

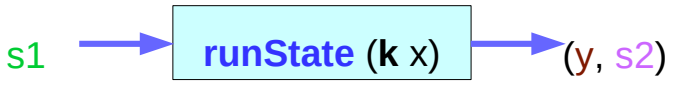


https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

Another implementation of >>=

```
instance Monad (State s) where

(>>=) :: State s a -> (a -> State s b) -> State s b
p  >>= k = state $ \ s0 ->
   let (x, s1) = runState p s0
   in runState (k x) s1
```

-- running the first processor on s0.



-- running the second processor on s1.

```
state ( \ s0 ->   (y, s2) )
```

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

References

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