State Monad Example (3G)

Young Won Lim
2/14/18

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

This document was produced by using LibreOffice.

Young Won Lim
2/14/18

Based on

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

Setting the State : **put**

put :: s **->** **State** s a

put newSt = **state** **$** _ -> ((), newSt)

Given a wanted state newState,

put generates a **state processor**
- ignores whatever the state it receives,
- updates the state to newState
- doesn't care about the result of this processor

- all we want to do is to change the state
- the tuple will be ((), newState)
- () : the **universal placeholder value**.

unwrapped state processor
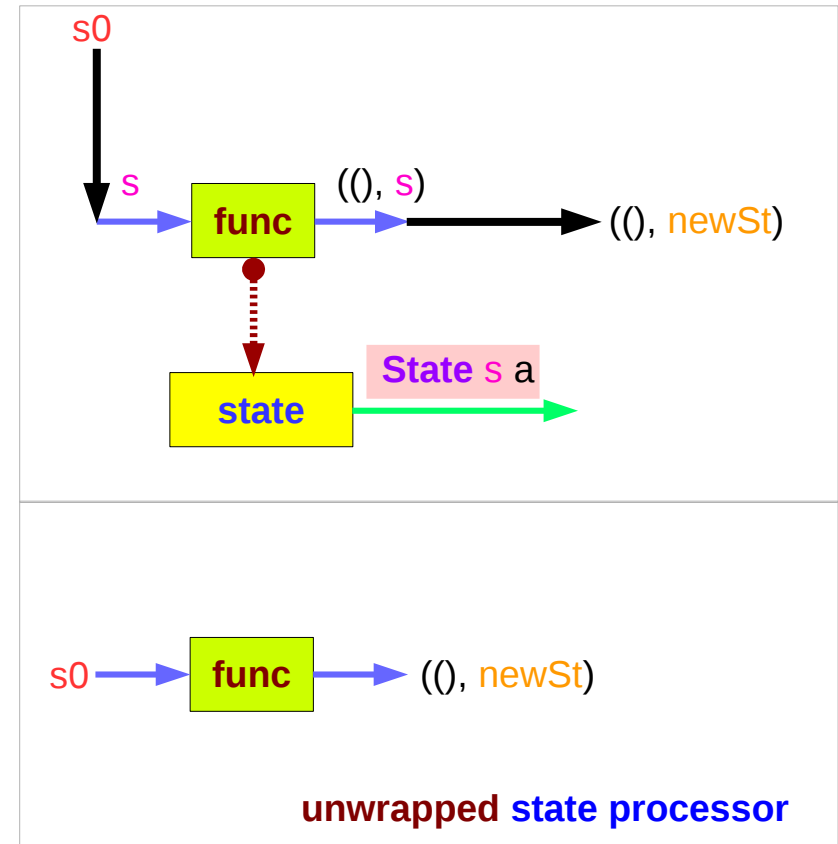
https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

Getting the State : **get**

get :: **State** s s

get = **state** $ \s -> (s, s)

get <u>generates</u> a **state processor**

- gives back the state s0
- as a result and as an updated state – (s0, s0)

- the state will remain <u>unchanged</u>
- a <u>copy</u> of the state will be made available through the result returned

unwrapped state processor
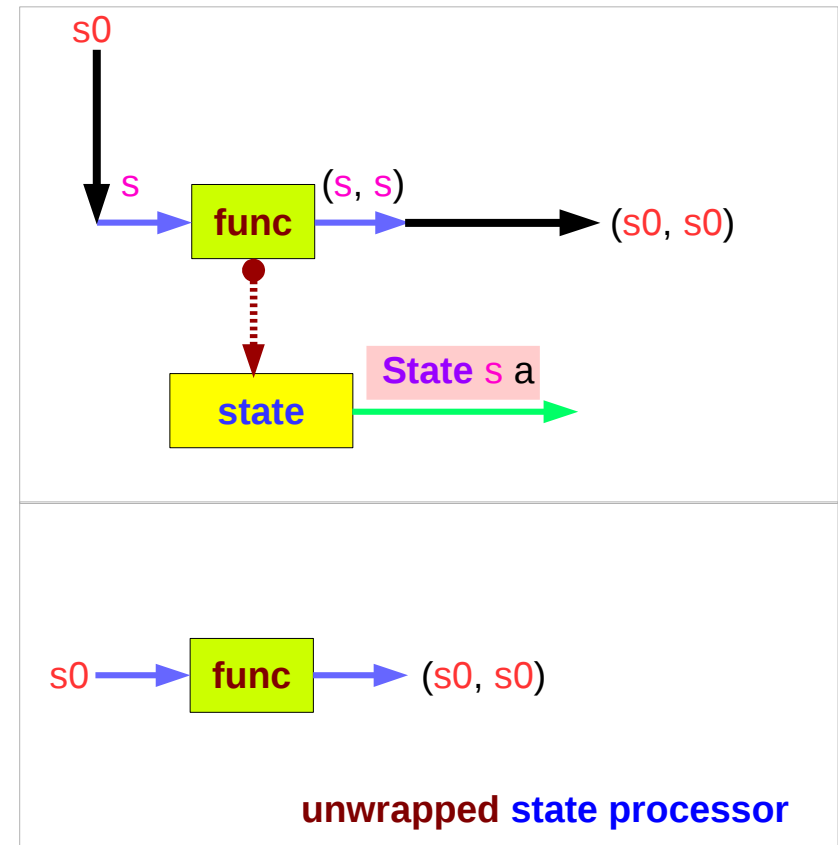
Example Codes

```
import Control.Monad.Trans.State

runState get 1
(1,1)
runState (return 'X') 1
('X',1)
runState get 1
(1,1)
runState (put 5) 1
((),5)


let postincrement = do { x <- get; put (x+1); return x }
runState postincrement 1
(1,2)
let predecrement = do { x <- get; put (x-1); get }
 runState predecrement 1
(0,0)
```

```
runState (modify (+1)) 1
((),2)
runState (gets (+1)) 1
(2,1)
evalState (gets (+1)) 1
2
execState (gets (+1)) 1
1
```

https://wiki.haskell.org/State_Monad

State Monad Example (3G)

6

Young Won Lim
2/14/18

Setting and Getting the State

put :: s **-> State** s a

put s **:: State** s a

put newState = **state $** _ -> ((), newState)

-- setting a state to newState

-- regardless of the old state

-- setting the result to ()

get :: State s s

get = **state $** \s -> (s, s)

-- getting the current state s

-- also setting the result to s
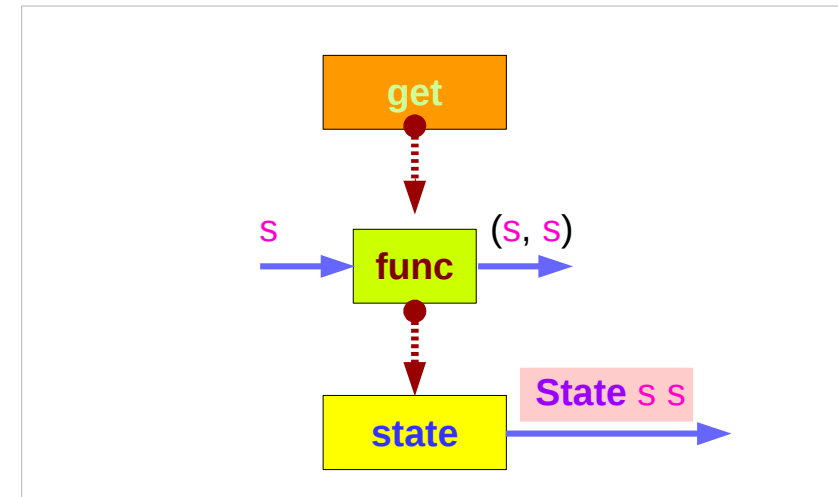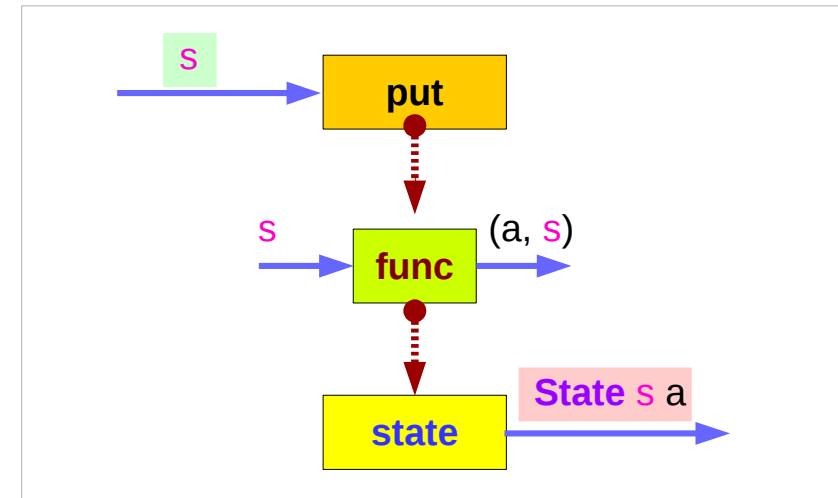
https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

Types and Values of **put** and **get**
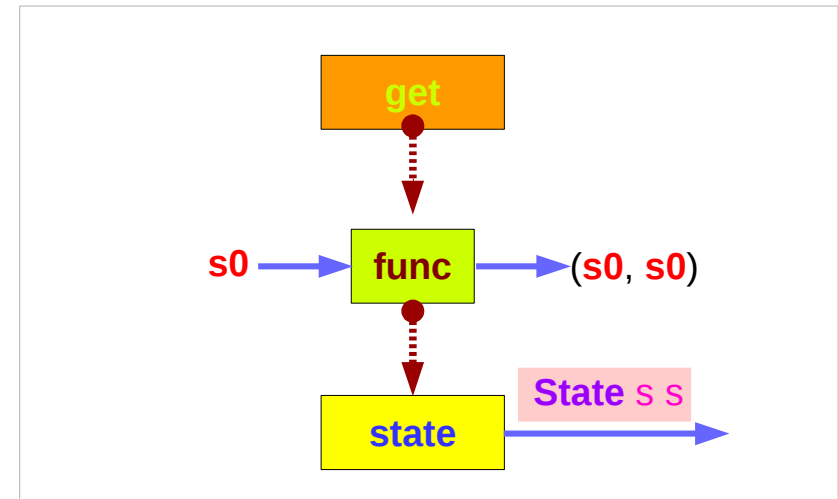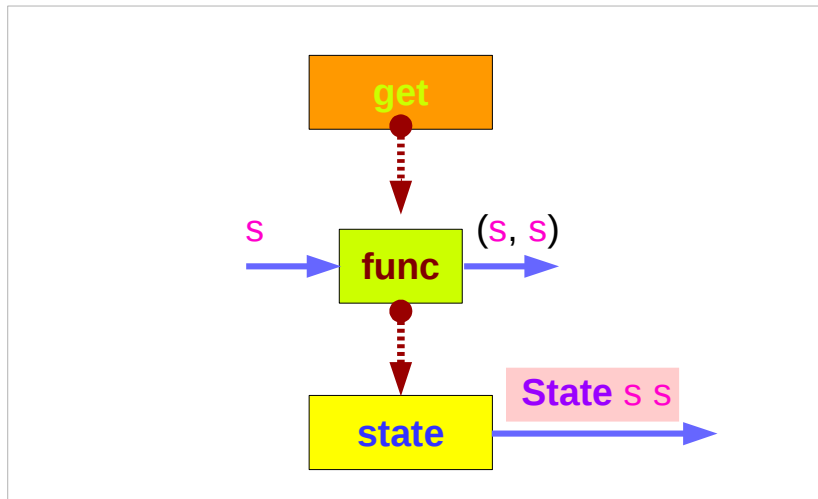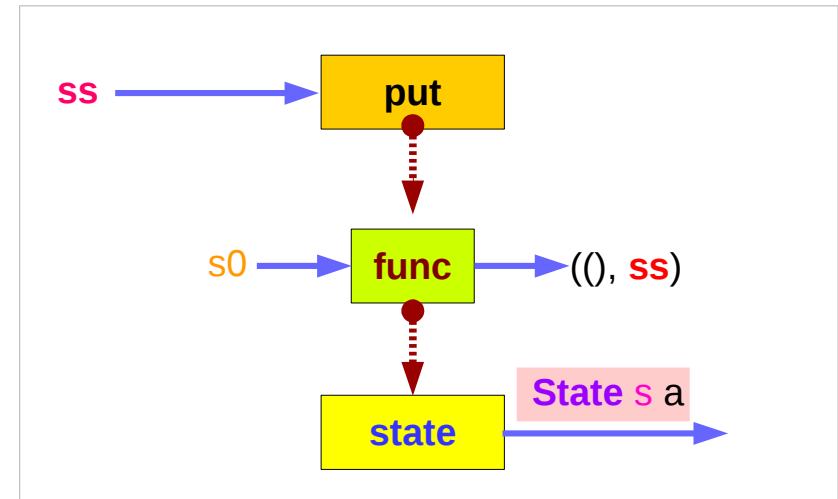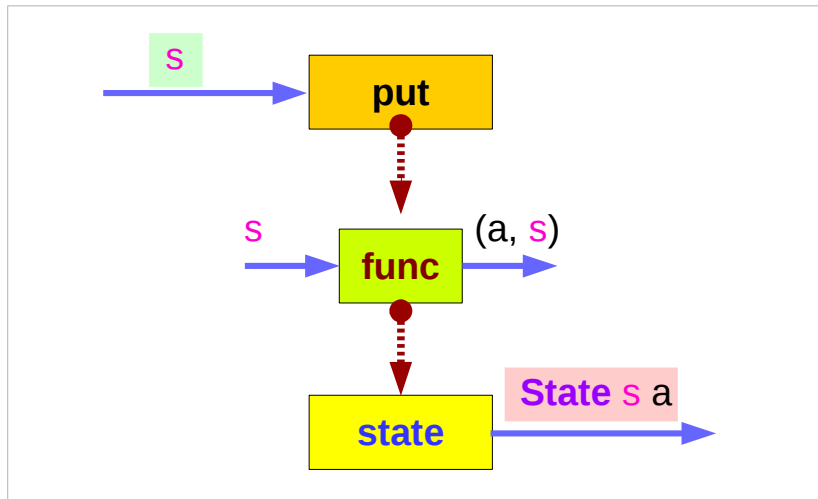
wrapped version of **put** and **get**

put :: s **-> State** s a

put s **:: State** s a

put newSt = **state $** _ -> ((), newSt)
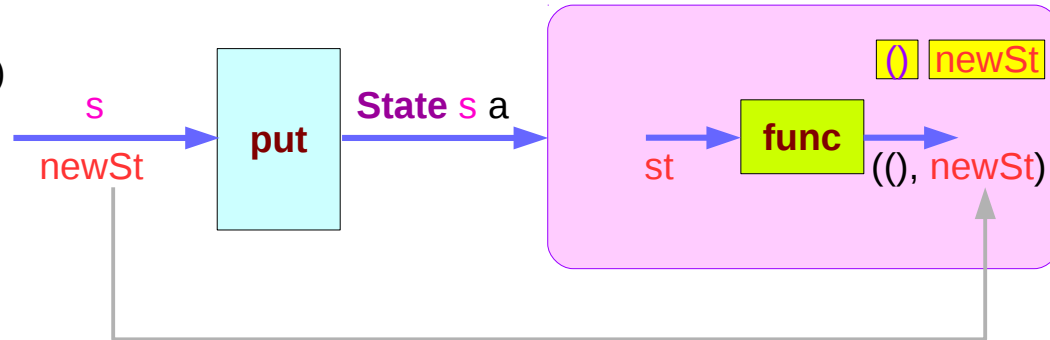
get :: State s s

get = **state $** \s -> (s, s)

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

3G
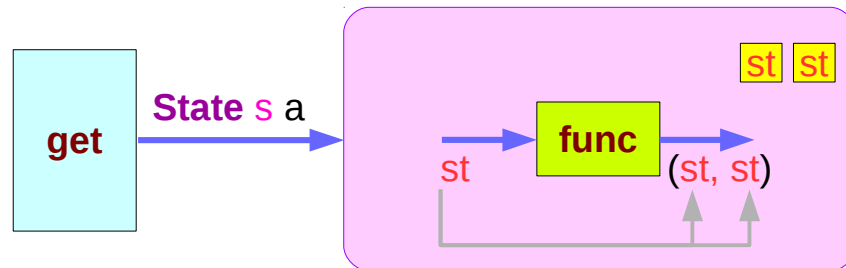
Executing the state processor

put :: s **-> State** s a

put newSt = **state $** _ -> ((), newSt)

runState (**put** newSt) s0 ➜ ((), newSt)

s0

func

st   ((), newSt)

((), newSt)
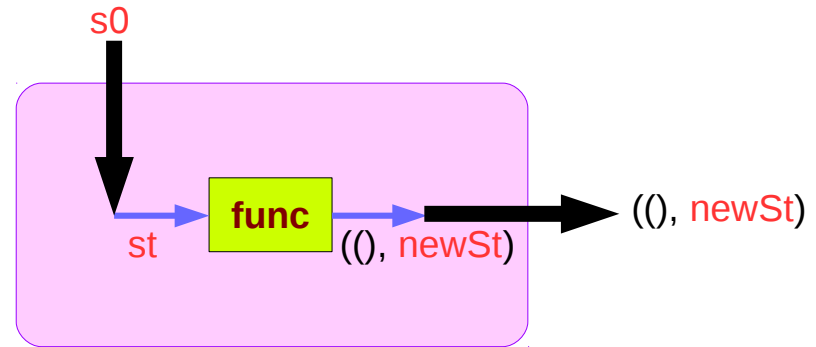
applying the function

---

get :: State s s

get = **state $** \s -> (s, s)

runState (**get**) s0 ➜ (s0, s0)

s0   **p** :: **State** s a

func

st   (st, st)

(s0, s0)
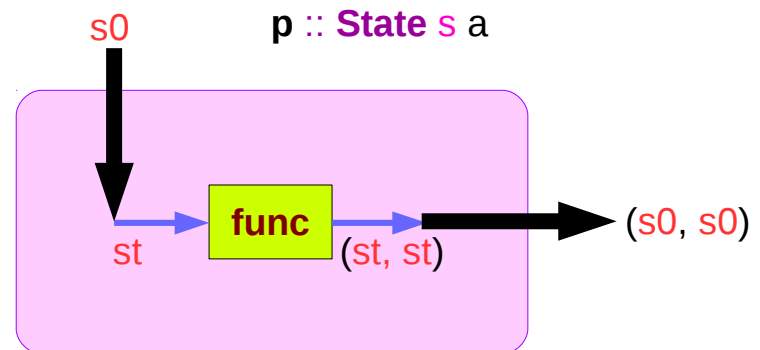
applying the function

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

State Monad Examples – **put**

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
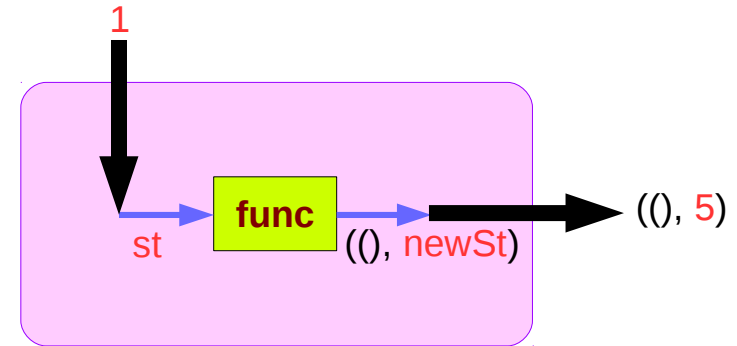
put :: s **-> State** s a
put newState = **state** $ _ -> ((), newState)

State Monad Examples – **get**

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
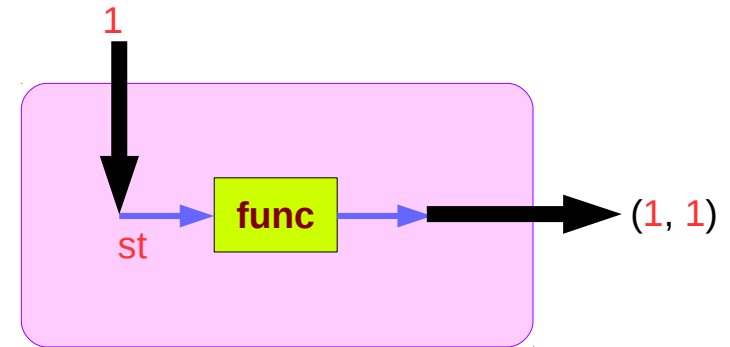
get :: State s s

get = **state** $ \s -> (s, s)

https://wiki.haskell.org/State_Monad

Think an unwrapped state processor

(**return** 5) ➜ `1 -> (5,1)` -- a way of thinking

get ➜ `1 -> (1,1)` -- a way of thinking

(**put** 5) ➜ `1 -> ((),5)` -- a way of thinking

Think an **unwrapped** state processor

---

a value of type (**State** s a) is

a **function** <u>from</u> initial state s

<u>to</u> final value a and final state s: (a,s).

these are usually wrapped,

but shown here unwrapped for simplicity.

(**return** 5) ➜ **state**(`1 -> (5,1)`) -- an actual way

get ➜ **state**(`1 -> (1,1)`) -- an actual way

(**put** 5) ➜ **state**(`1 -> ((),5)`) -- an actual way

wrapping the state processor

https://wiki.haskell.org/State_Monad

State Monad Examples – **return**, **get**, and **put**

Return leaves the state unchanged and sets the result:

-- ie: (**return** 5) ➔ 1 -> (5,1) -- a way of thinking

runState (**return** 5) 1 ➔ (5,1)

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

-- ie: **get** ➔ 1 -> (1,1) -- a way of thinking

runState get 1 ➔ (1,1)

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

-- ie: (**put** 5) ➔ 1 -> ((),5) -- a way of thinking

runState (**put** 5) 1 ➔ ((),5)

https://wiki.haskell.org/State_Monad

State Monad Examples – **modify** and **gets**

```
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) }
```

```
runState (modify (+1)) 1                    (+1) 1 → 2 :: s
   ➤    ((),2)


runState (gets (+1)) 1                       (+1) 1 → 2 :: a
   ➤    (2,1)
```

```
evalState (gets (+1)) 1                       → :: s state
   ➤    2


execState (gets (+1)) 1                       → :: a result
   ➤    1
```

https://wiki.haskell.org/State_Monad

```
x <- get; put (f x)
x <- get; return (f x)
```

- inside a monad instance
- unwrapped implementations of **modify** and **gets**

Unwrapped Implementation – **return**, **get**, and **put**

Return leaves the state unchanged and sets the result:

-- ie: (**return** 5)   ➔   1 -> (5,1)   -- a way of thinking

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)   -- a way of thinking

get :: **State** s s

get s = (s,s)

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

-- ie: (**put** 5)   ➔   1 -> ((),5)   -- a way of thinking

put :: s -> **State** s ()

put x s = ((),x)

https://wiki.haskell.org/State_Monad

(x,s)

(s,s)

((),x)

- inside a monad instance
- unwrapped implementations
  of **return**, **get**, and **put**

The Result of a Stateful Computation

a **stateful computation** is a **function** that

   takes some **state** and

   returns a **value** along with some **new state**.

That function would have the following type:

s -> (a,s)

s is the type of the **state** and

a the **result** of the **stateful computation**.

s -> (a, s)

s → ☐ → (a, s)
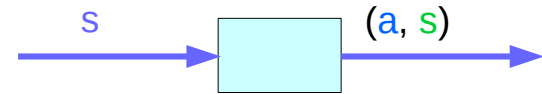
a <u>function</u> is an executable <u>data</u>

when <u>executed</u>, a <u>result</u> is produced

action, execution, result

$$s \rightarrow (a, s)$$

Stateful Computations Inside the State Monad

inside a monad,

when **sc** is a **s**tateful **c**omputation

then the result of the stateful computation **sc**

can be assigned to x

sc :: State s a

x <- **sc**

x :: a   (the execution result of **sc**)

$$s \rightarrow (a, s)$$

the result type

https://stackoverflow.com/questions/11250328/working-with-the-state-monad-in-haskell

get inside the State Monad
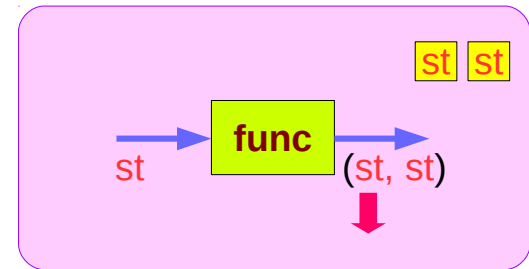
inside the **State** monad,

get returns the current monad instance

whose type is **State** s a

`x <- get`

the stateful computation is performed

over the current monad instance returned by **get**

the result of the stateful computation is st::s

thus x will get the st

x :: a   the execution result of **get**

st st

st → func → (st, st)

Getting the current state inside the State Monad

inside the **State** monad,

get returns the <u>current</u> <u>monad</u> <u>instance</u>

whose type is **State** s a

to get the <u>current</u> <u>state</u> st, do

```
s <- get
```

s will have the value of the <u>current</u> <u>state</u> st

this is like **evalState** is called with the current monad instance

- **get**
- **current monad instance**
- **stateful computation**
- **result :: s**

https://stackoverflow.com/questions/11250328/working-with-the-state-monad-in-haskell
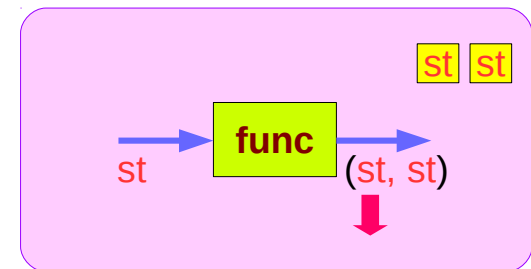
put and **get** inside State Monad

put :: s **-> State** s a

put newSt = **state** $ _ -> ((), newSt)

put :: s **->** ()

the result type :: ()

stateful computation of **put**
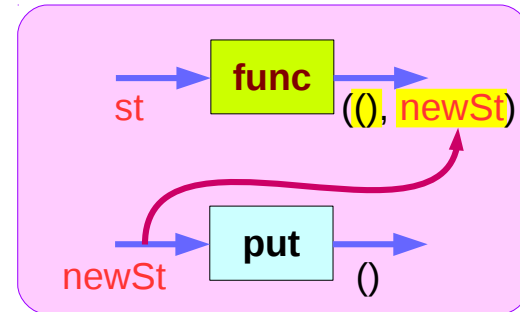
get :: State s s

get = **state** $ \s -> (s, s)

get :: s

the result type :: s

stateful computation of **get**
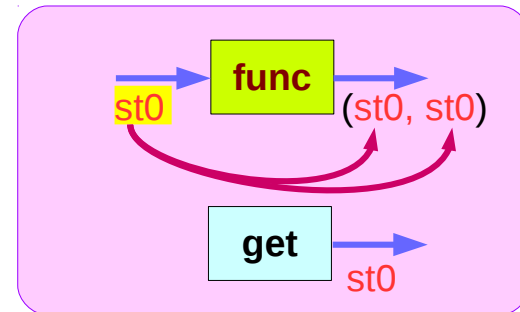
https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

Inside Functions and runState Functions

Most monads are equipped with some "*run*" functions

such as **runState**, **execState**, and so forth.

But, frequent calling such <u>functions</u> <u>inside</u> <u>the</u> <u>monad</u>

   shows that the functionality of the monad does <u>not</u> <u>fully</u> <u>exploited</u>

s0 <- **get**   -- <u>read</u> the <u>state</u> of the current instance   **let p = state** (\y -> (y, y+1))

let (a,s1) = **runState p** s0   -- pass the <u>state</u> to **p**, <u>get</u> <u>new</u> <u>state</u>

put s1   -- <u>save</u> <u>new</u> state

a <- **p**   -- the stateful computation **p** <u>updates</u> the state to s1

                               -- the <u>result</u> of the state returned is assigned to a

https://stackoverflow.com/questions/11250328/working-with-the-state-monad-in-haskell

Redundant computation examples (1)

s0 <- **get**
let (a,s1) = **runState p** s0
put s1

the same binding variable (a)
the same state (s1)

a <- **p**

State s a | s | (a, s) |
| p | s0 | **runState** |

(a, s1)
(a, s1)

func
s0 → (s0, s0)

get
s0

s0

the current monad <u>instance</u>
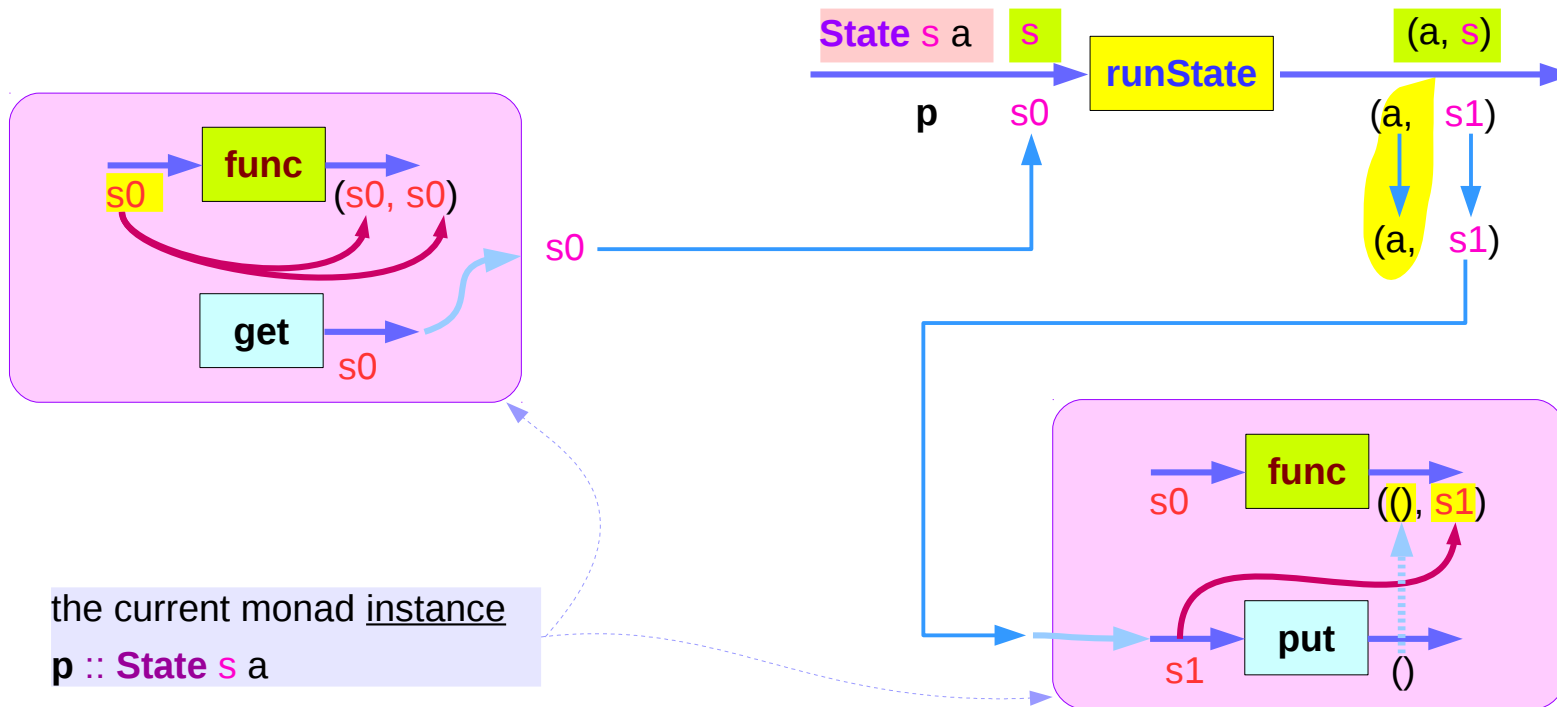**p** :: **State** s a

func
s0 → ((), s1)

put
s1 → ()

https://stackoverflow.com/questions/11250328/working-with-the-state-monad-in-haskell

Redundant computation examples (2)

a <- **p**

-- the stateful computation **p** <u>updates</u> the state to s1
-- the <u>result</u> of the state returned is assigned to a

p :: **State** s a

stateful computation **p**   return the result a

https://stackoverflow.com/questions/11250328/working-with-the-state-monad-in-haskell

Inside Functions and runState Functions

```
collectUntil f comp = do
    st <- get                              -- Get the current state
    if f st then return [ ]                -- If it satisfies predicate, return
        else do                            -- Otherwise...
            x  <- comp                     -- Perform the computation s
            xs <- collectUntil f comp      -- Perform the rest of the computation
            return (x:xs)                  -- Collect the results and return them
```

comp :: State s a

st :: s

x :: a

xs :: [a]

simpleState = state (\x -> (x,x+1))

simpleState :: State s a

*Main> evalState (collectUntil (>10) simpleState) 0
[0,1,2,3,4,5,6,7,8,9,10]

a s

s → func → (a, s)

https://stackoverflow.com/questions/11250328/working-with-the-state-monad-in-haskell
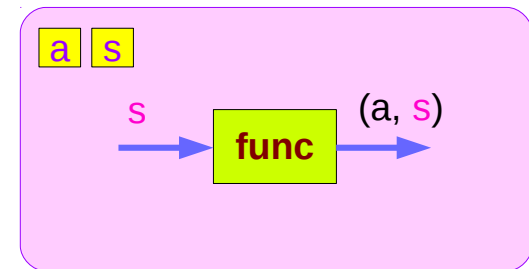
Inside Functions and runState Functions

```
collectUntil f comp = do
    st <- get
    if f st then return [ ]
        else do
            x  <- comp    -- stateful computation
            xs <- collectUntil f comp
            return (x:xs)
```
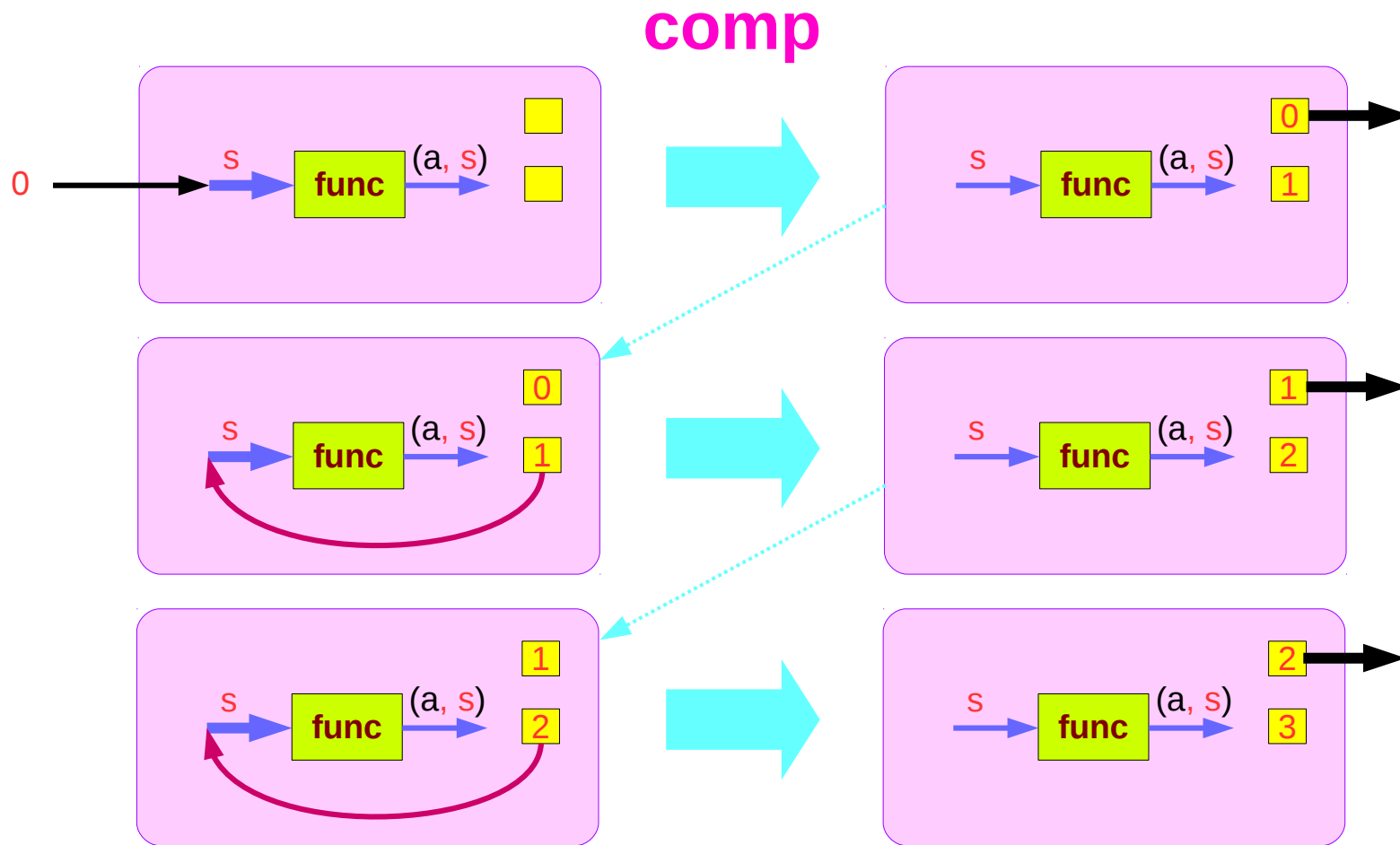
```
simpleState = state (\x -> (x,x+1))
```

get	st←0	comp :	0 → (0, 1)	x←0
get	st←1	comp :	1 → (1, 2)	x←1
get	st←2	comp :	2 → (2, 3)	x←2
get	st←3	comp :	3 → (3, 4)	x←3
get	st←4	comp :	4 → (4, 5)	x←4
get	st←5	comp :	5 → (5, 6)	x←5
get	st←6	comp :	6 → (6, 7)	x←6
get	st←7	comp :	7 → (7, 8)	x←7
get	st←8	comp :	8 → (8, 9)	x←8
get	st←9	comp :	9 → (9, 10)	x←9
get	st←10	comp :	10→(10, 11)	x←10

```
*Main> evalState (collectUntil (>10) simpleState) 0
[0,1,2,3,4,5,6,7,8,9,10]
```
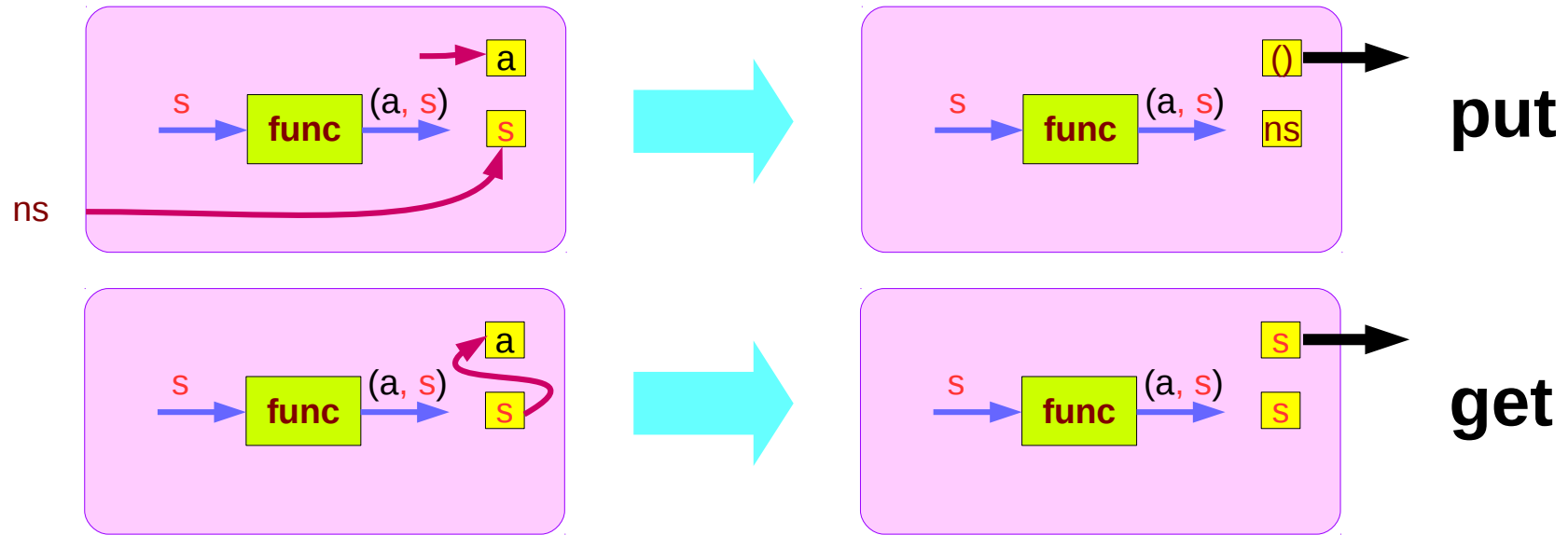
https://stackoverflow.com/questions/11250328/working-with-the-state-monad-in-haskell

Stateful Computation of **comp**

comp

Stateful Computations of **put** & **get**

Inside Functions and runState Functions

collectUntil :: MonadState t m => (t -> Bool) -> m a -> m [a]

collectUntil :: (t -> Bool) -> State t a -> State t [a]

Inside function examples

collectUntil :: (s -> Bool) -> **State** s a -> **State** s [a]

collectUntil f s = **step**

  where

    step = **do** a <- s

          liftM (a:) **continue**

    continue = **do** s' <- **get**

            if **f** s' then **return** [] else **step**

simpleState = **state** (\x -> (x,x+1))

*Main> **evalState** (**collectUntil** (>10) **simpleState**) 0

[0,1,2,3,4,5,6,7,8,9,10]

https://stackoverflow.com/questions/11250328/working-with-the-state-monad-in-haskell

liftM

liftM :: (**Monad** m) => (a -> b) -> m a -> m b

mapM :: (**Monad** m) => (a -> m b) -> [a] -> m [b]

liftM lifts a function of type a -> b to a monadic counterpart.

mapM applies a function which yields a monadic value to a list of values,

yielding list of results embedded in the monad.

> **liftM** (**map toUpper**) **getLine**

Hallo

"HALLO"

> :t **mapM** return "monad"

mapM return "monad" :: (**Monad** m) => m [Char]

https://stackoverflow.com/questions/5856709/what-is-the-difference-between-liftm-and-mapm-in-haskell

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)

```haskell
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

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)

randomR

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

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

assume a is Int   (a, a) : range
and g is StdGen   a seed

A seed of the type **StdGen**
A new seed is generated
by **newStdGen**

the type of **randomR**

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

  (**Int**, **StdGen**)
  (**a random value, a new seed**)

already have a **state processing function**

randomR

If you choose to take a <u>seed</u>,

it should be of type **StdGen**, and you can use **randomR**

to generate a number from it.

Use **newStdGen** to create a new seed

(this will have to be done in IO).

> **import System.Random**

> **g <- newStdGen**

> **randomR (1, 10) g**

(1,1012529354 2147442707)

The result of randomR is a tuple

(a **random value**, a **new seed)**

A seed of the type **StdGen**

A new seed is generated

by **newStdGen**

randomR

Otherwise, you can use randomRIO

to get a random number directly in the IO monad,

with all the StdGen stuff taken care of for you:

> import System.Random
> randomRIO (1, 10)
6

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

randomR

```
import Control.Monad.Trans.State
import System.Random

-- The StdGen type we are using is an instance of RandomGen.
randomR :: (Random a, RandomGen g) => (a, a) -> g -> (a, g)

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

randomR

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


rollDie :: State StdGen Int
rollDie = do generator <- get
             let (value, newGenerator) = randomR (1,6) generator
             put newGenerator
             return value




GHCi> evalState rollDie (mkStdGen 0)
6
```

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

randomR

```
rollDice :: State StdGen (Int, Int)
rollDice = liftA2 (,) rollDie rollDie

GHCi> evalState rollDice (mkStdGen 666)
 (6,1)
```

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