Monad P1 : Monadic Operations (3A)

Young Won Lim
3/9/19

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

Monadic Effect

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

https://en.wikibooks.org/wiki/Haskell/Understanding_monads/IO
https://stackoverflow.com/questions/2488646/why-are-side-effects-modeled-as-monads-in-haskell
https://stackoverflow.com/questions/7840126/why-monads-how-does-it-resolve-side-effects
https://www.cs.hmc.edu/~adavidso/monads.pdf

Monad Applications

1. **Exception Handling**	**Maybe a**
2. **Accumulate States**	**State s a**
3. **IO Monad**	**IO a**

http://www.idryman.org/blog/2014/01/23/yet-another-monad-tutorial/

Monadic Operations – a function form

Monadic operations type signature

val-in-type-1 **->** ... **->** val-in-type-n **->** **effect-monad** val-out-type

the types of **inputs** to a **monadic operation**

the type of a **return** **mondic** **value** from a **monadic operation**

function type

https://stackoverflow.com/questions/16892570/what-is-in-haskell-exactly

Monadic Operations – returning a monadic value

Monadic operations type signature

val-in-type-1 **->** ... **->** val-in-type-n **->** **effect-monad** val-out-type

a monadic operation

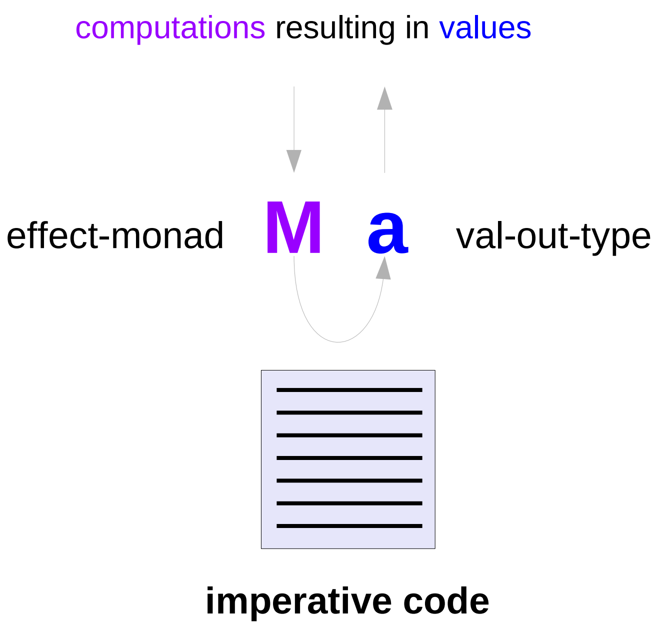
= a function

- **inputs**
- **a return monadic value**

– returns a **function** as a **value**

– **effect monad**

– evaluating this returns **val-out-type**



https://stackoverflow.com/questions/16892570/what-is-in-haskell-exactly

Monadic Operations – the result of a monadic value

Monadic operations type signature

val-in-type-1 **->** ... **->** val-in-type-n **->** **effect-monad** val-out-type

effect-monad produces a

result of a type of val-out-type

actions

computations

statements

in the imperative language

https://stackoverflow.com/questions/16892570/what-is-in-haskell-exactly

Monadic Operations – type application

		function	argument
val-in-type-1 **->** ... **->** val-in-type-n	**->**	**effect-monad**	val-out-type

type application

cf) function application

Maybe a → **Maybe Int**

f x **f 3**

the return type is a **type application** like a function application

the function part tells you ………………………… **effect-monad**

which effects are possible

the argument part tells you ……………………… val-out-type

what sort of value is produced by the operation.

https://stackoverflow.com/questions/16892570/what-is-in-haskell-exactly

Parametric Polymorphism

when the **type** of a **value** contains
one or more (unconstrained) **type variables**,
so that the **value** may adopt any **type**
that results from substituting those variables with **concrete types**.

any type in which a **type variable**,
denoted by a type name beginning with a **lowercase letter**,
can appear without **constraints** (i.e. no left **=>**)

id :: a -> a
contains an unconstrained **type variable a** in its type,

Char -> Char
Integer -> Integer
(Bool -> Maybe Bool) -> (Bool -> Maybe Bool)

Maybe a

x :: Maybe Int

y :: Maybe String

https://wiki.haskell.org/Polymorphism

Function Application

passing an **argument** to the **function**

f :: Int -> Int
x :: Int

f x :: Int is an expression
where the **expression x** is applied as an **argument to f ***.

$ is often explained as the **function application operator**,
since **f $ x = f x** is more-or-less its definition

Applying a function is the same as **calling** it,
by supplying an **argument**.

```
-- A function
f :: a -> a
f x = x

-- Application of f
f 100
```

https://stackoverflow.com/questions/52058692/the-term-function-application-in-haskell

Type Annotation vs TypeApplication

Type Annotation

```
Prelude> id "a"
"a"
Prelude> id (3 :: Int)
3
```

TypeApplications extension allows explicit type arguments.

```
Prelude> :set -XtypeApplications

Prelude> id @String "a"
"a"
Prelude> id @Int 3
3
```

https://ghc.haskell.org/trac/ghc/wiki/TypeApplication

TypeApplication

a feature that lets a programmer explicitly declare
what types should be instantiated
for the **arguments** to a **function application**,
in which the function is **polymorphic**
(containing **type variables** and possibly **constraints**) .

Doing so essentially expedite
the **type variable unification** process,
which is what GHC normally attempts when dealing
with **polymorphic function application**.

:set -XTypeApplications

```
answer_read = show (read @Int "3")              -- "3" :: String
answer_show = show @Integer (read "5")          -- "5" :: String
answer_showread = show @Int (read @Int "7")     -- "7" :: String
```

https://ghc.haskell.org/trac/ghc/wiki/TypeApplication

Monadic Operations – **IO** and **State** Monads

val-in-type-1 **->** ... **->** val-in-type-n **->** **effect-monad** val-out-type

Monadic operations such as **IO** or **State**

have a **return value**, as well as

performing **side-effects**.

the only purpose of using these monadic operations is

to perform a **side-effect**,

writing to the screen .... IO Monad

storing some state .... State Monad

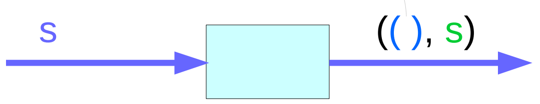
https://stackoverflow.com/questions/16892570/what-is-in-haskell-exactly

Monadic Operation – the result type

val-in-type-1 **->** ... **->** val-in-type-n **->** **effect-monad** val-out-type

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

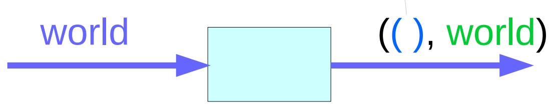
result



the execution result type of
the returned function

putStr :: String -> IO ( )

result



https://stackoverflow.com/questions/16892570/what-is-in-haskell-exactly

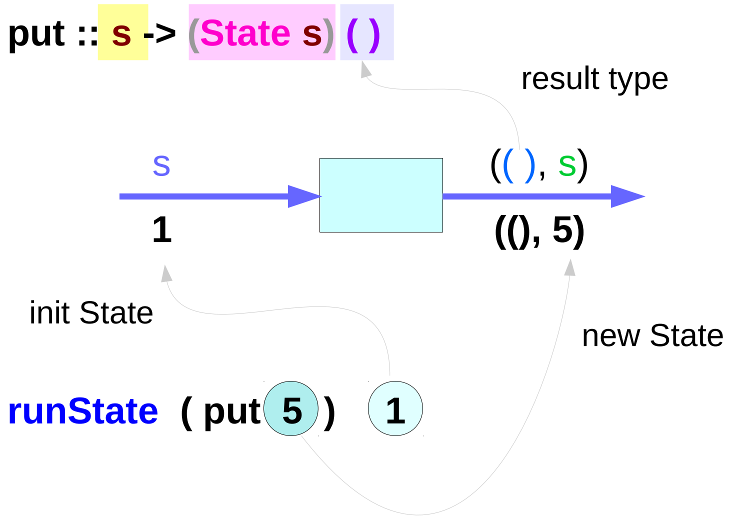
Monadic Operations – **put** example

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

the operation is used *only for its effect*;
the *value* delivered is *uninteresting*

one value input type — **s**
the **effect-monad** — **State s**
the value output type — **( )**

effect-monad	val-out-type
(State s)	( )



https://stackoverflow.com/questions/16892570/what-is-in-haskell-exactly

Monadic Operations – **putStr** example

putStr :: String -> IO ( )

delivers a string to **stdout**

but does not return anything meaningful

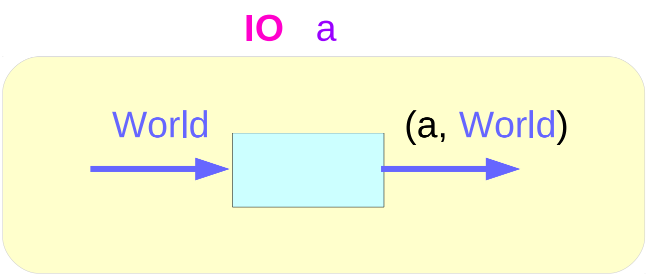
( ) val-out-type

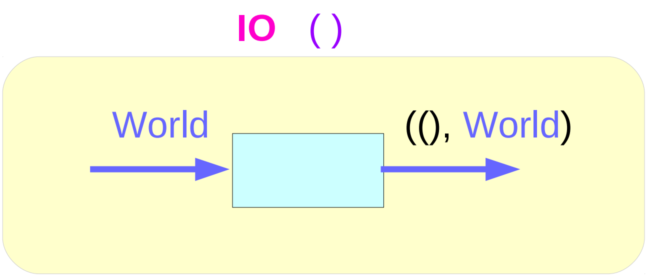
one value input type **s**

the **effect-monad** **IO**

the value output type **( )**

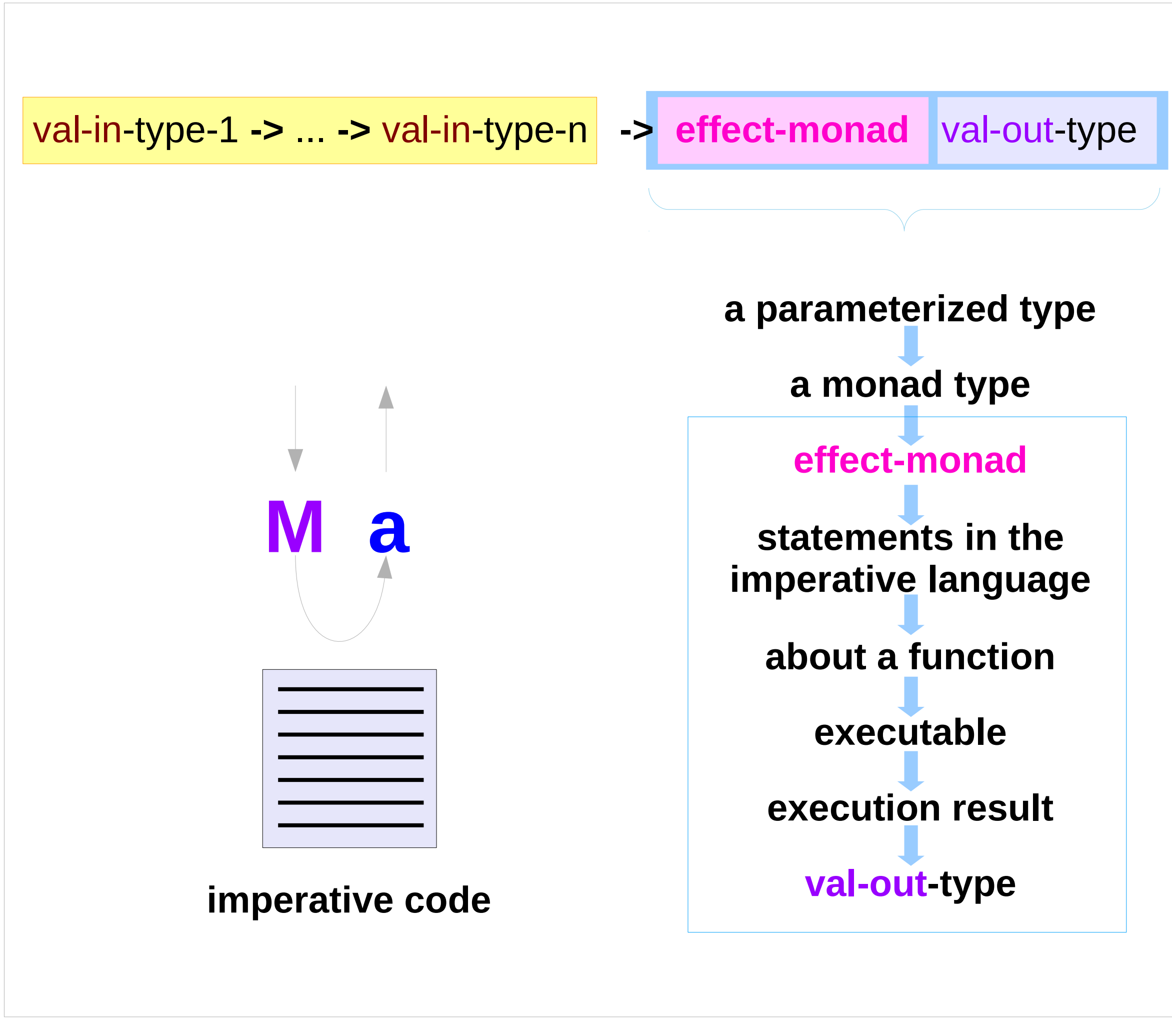
effect-monad	val-out-type
IO	( )

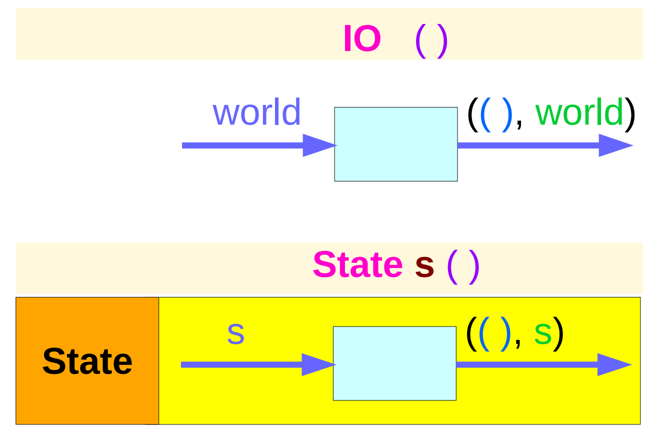




https://stackoverflow.com/questions/16892570/what-is-in-haskell-exactly

Monadic Operations – underlying functions

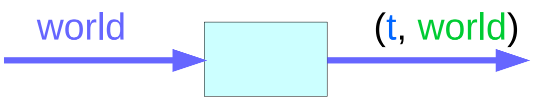




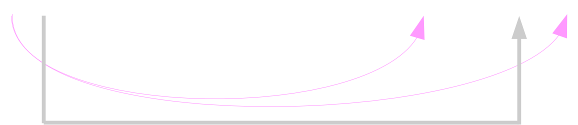
https://stackoverflow.com/questions/16892570/what-is-in-haskell-exactly

IO t and State s a types

type **IO** t = World -> (t, World)   type synonym



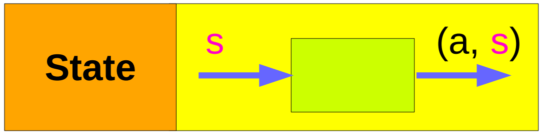
newtype State s a = **State { runState** :: s -> (a, s) **}**



s : the type of the state,

a : the type of the produced result

s -> (a, s) : function type



accessor function

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

Monadic Bind

We might not see the **hidden effects**,

but the compiler does.

The compiler de-sugars every **do** block and type-checks it.

The **state** might look like a **global variable** but it's not.

monadic bind makes sure that

- the **state** is threaded from function to function.
- it's never **shared**.
- in a concurrent Haskell code, there will be no **data races**.

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

M :: m a
F :: a -> m b
G :: b -> m c
H :: c -> m d
```

M >>= F >>= G >>= H

monadic operations
with a single input
can be chained

https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/12-State-Monad

Parameter Hiding

If you have a **global environment**,
that is accessed by various **functions**

A **global environment** may be initialized by a **configuration file**
then you should thread it as a parameter to your **functions**
after having set it up in your **main** action.

instead of using annoying explicit **parameter** passing,
you can use a **Monad** to hide it

configuration file

cumbersome parameter passing

Monad

parameter hiding

f :: Int -> World -> (Int, World) — **non-pure (side effects)**

IO a = World -> (a, World) — **pure**
f :: Int -> IO Int

https://wiki.haskell.org/Global_variables

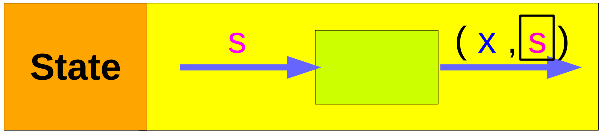
Global mutable variable in the **State** Monad

a **do** block looks very much like **imperative code**
with hidden **side effects**.

State monad code looks as if
the **state** were a **global mutable variable**.

- to access it, use **get** with no arguments
- to modify it, call **put** that returns no value

State Int Int



global mutable variable :: type s

https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/12-State-Monad

Stateful computations

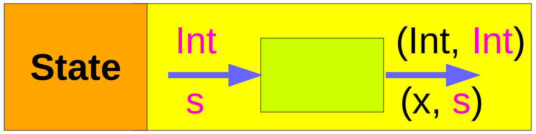
Haskell does not have **states**

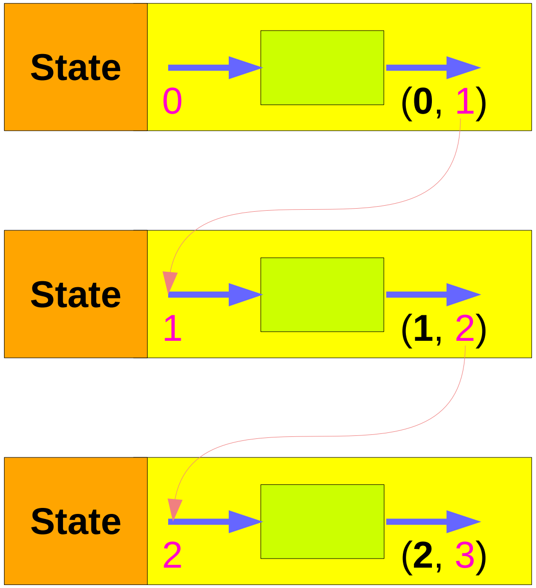
but it's type system is powerful enough

to construct the stateful program flow

function application enables

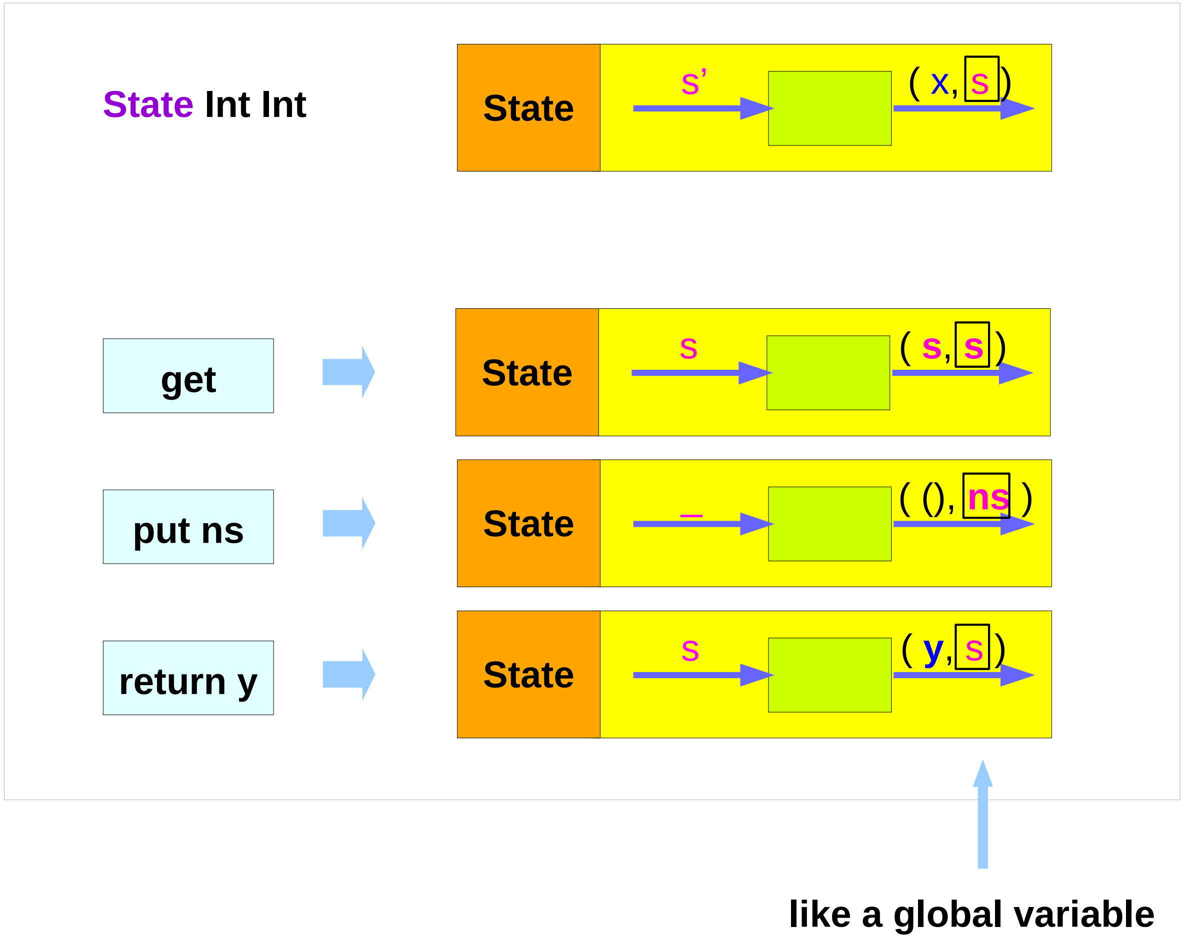
stateful computations





http://www.idryman.org/blog/2014/01/23/yet-another-monad-tutorial/

State Monad Methods



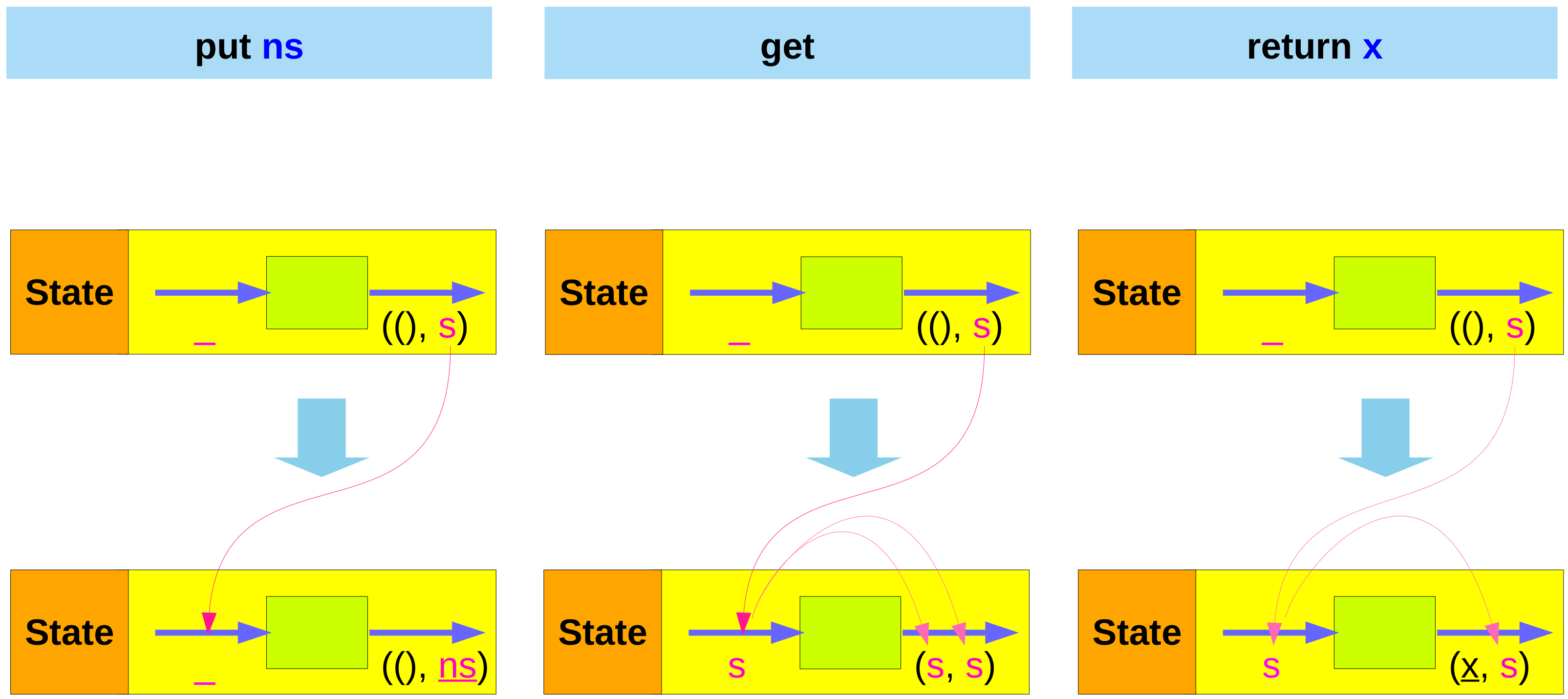
current monadic value

resulting monadic value by the **get** method — current state **s**

resulting monadic value by the **put ns** method — new state **ns**

resulting monadic value by the **return y** method — result **y**

put, **get**, **return** methods summary



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

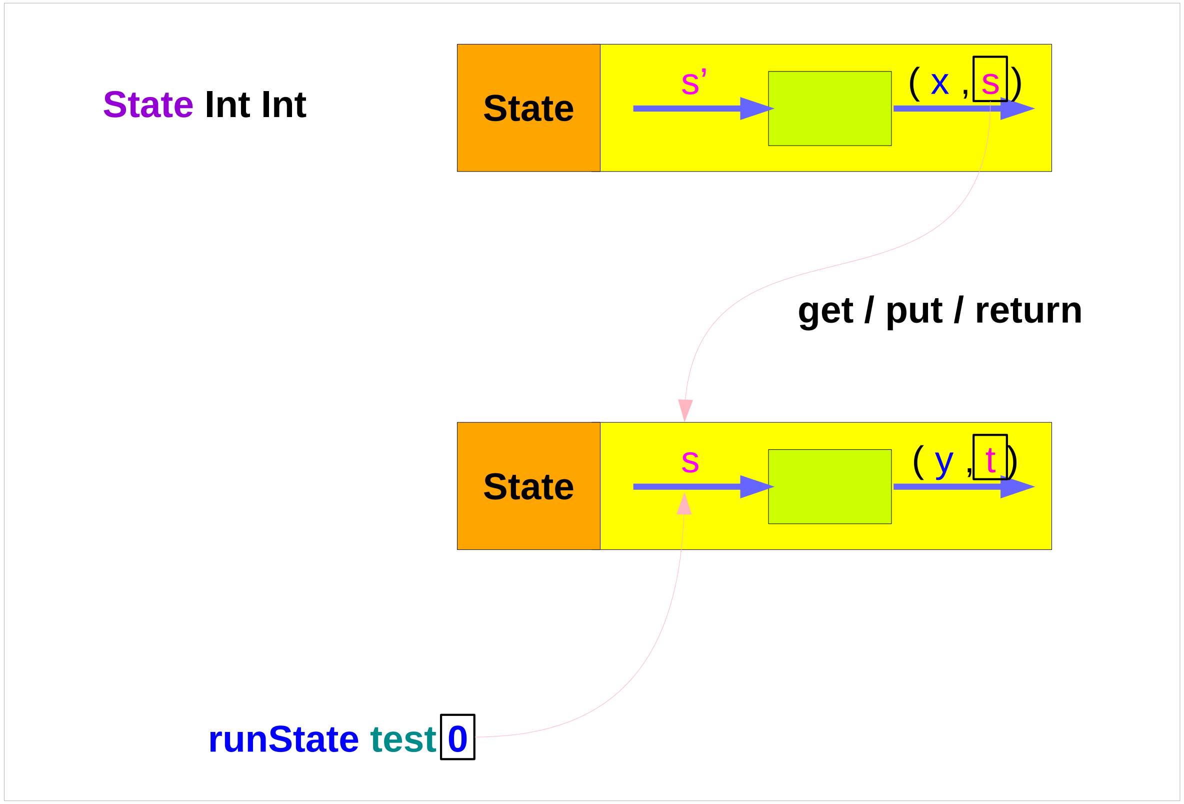
Global Variable Example

```haskell
import Control.Monad.Trans.State

tick :: State Int Int
tick = do  n <- get             -- read Int state
           put (n+1)            -- write Int state
           return n


test  = do  tick                -- (0,1)
            tick                -- (1,2)


runState test 0                 -- (1,2)
```

```haskell
test  = tick >> tick
```

https://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-State-Lazy.html

Threading the state



current monadic value

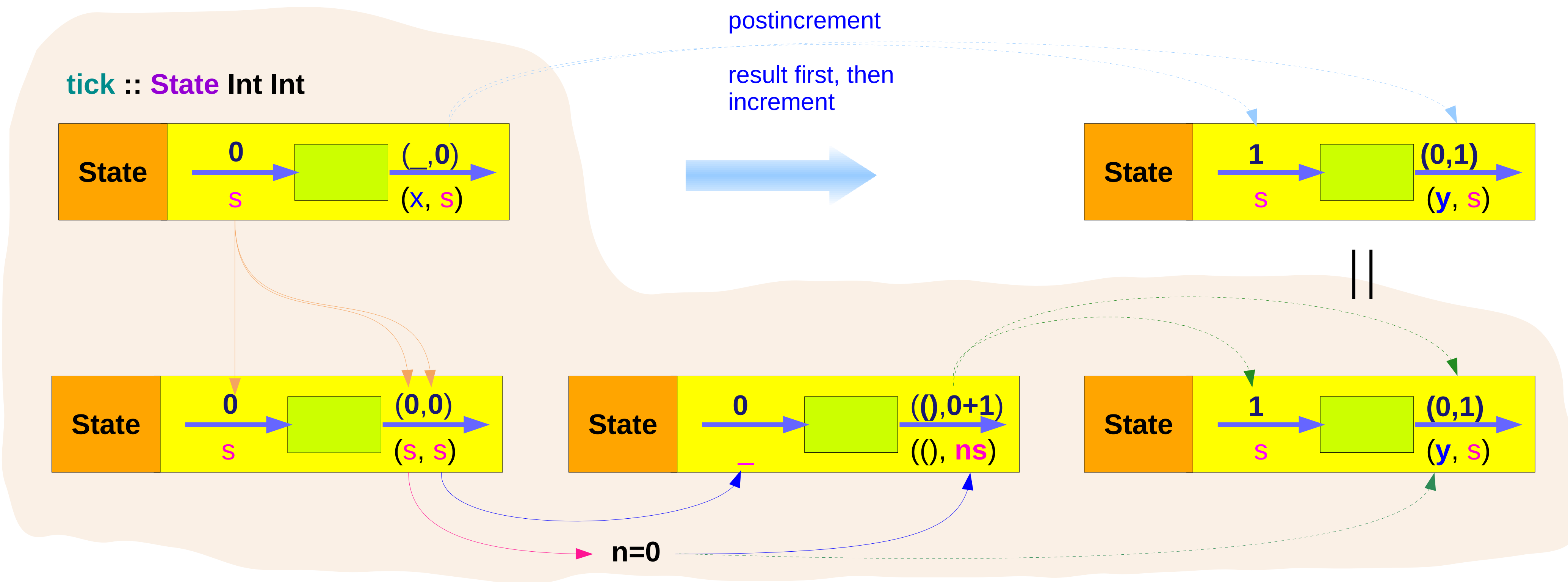
next monadic value via

1) get, put, return methods

2) runState

tick – State Monad Value

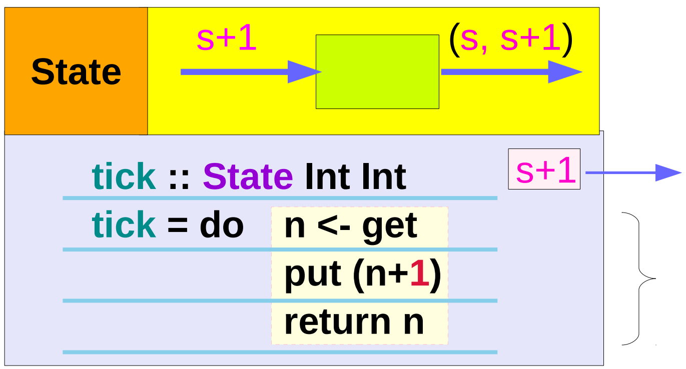
```
tick :: State Int Int
tick = do  n <- get          -- read Int state
           put (n+1)         -- write Int state
           return n
```



n <- get **put (n+1)** **return n**

Like a global mutable variable

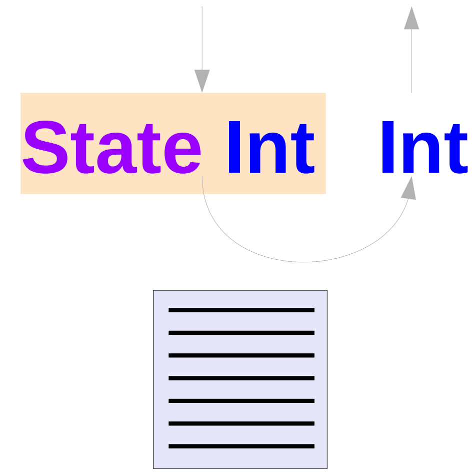
tick :: State Int Int



like a global variable

underlying operation

computations resulting in values

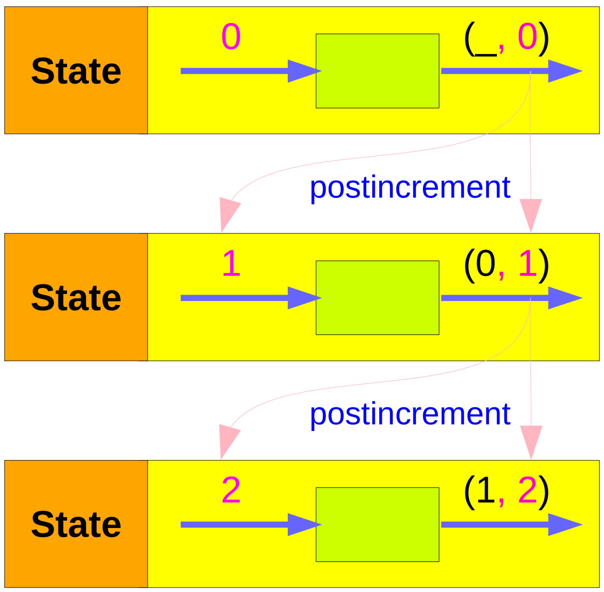


imperative code

a **do** block looks very much like **imperative code**
with hidden **side effects**.

State monad code looks as if
the **state** were a **global mutable variable**.

Evaluating **tick** twice (1)



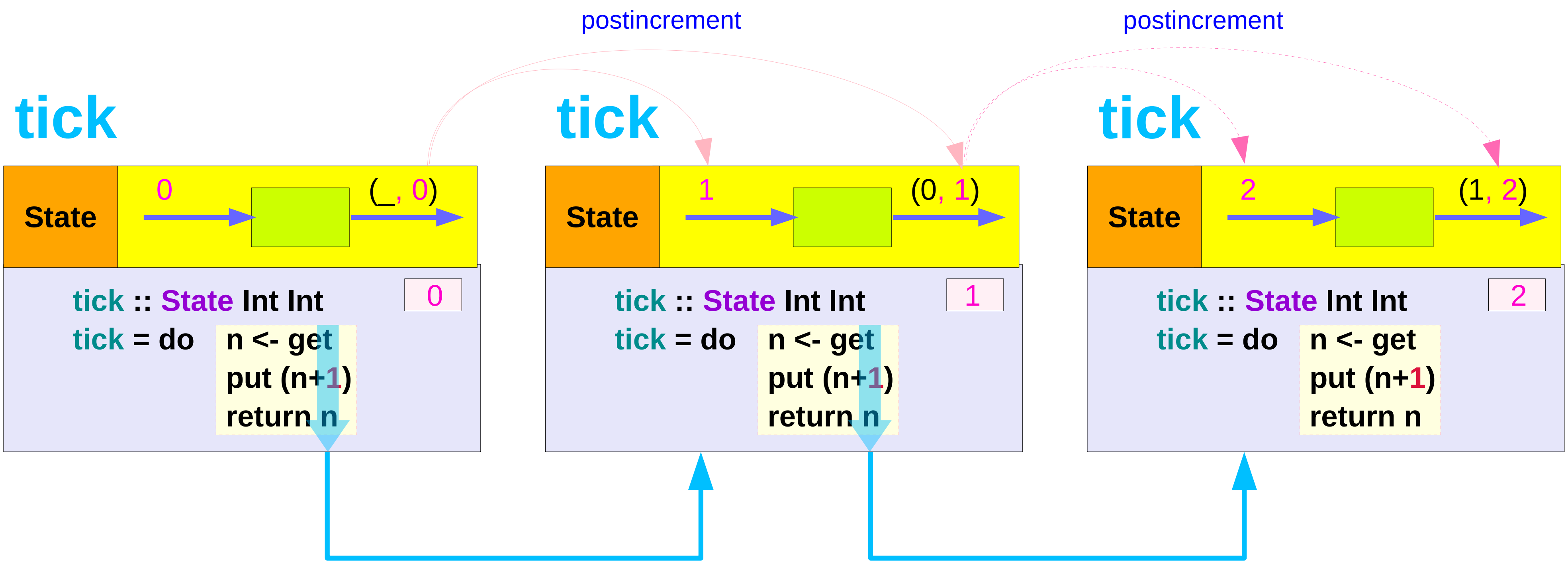
initial
tick

after the 1st evaluation
tick

after the 2nd evaluation
tick

```
import Control.Monad.Trans.State

tick :: State Int Int
tick = do   n <- get      -- read Int state
            put (n+1)     -- write Int state
            return n

test  = do  tick          -- (0,1)
            tick          -- (1,2)

runState test 0           -- (1,2)
```

Evaluating **tick** twice (2)



after the 1st evaluation of **tick**

after the 2nd evaluation of **tick**

Global Variable Example (1)

```
import Control.Monad.Trans.State

tick :: State Int Int
tick = do  n <- get              -- read Int state
           put (n+1)             -- write Int state
           return n

tick2 :: State Int Int
tick2 = do n <- get              -- read Int state
           put (n+2)             -- write Int state
           return n

test  = do tick                  -- (0,1)
           tick                  -- (1,2)
           tick2                 -- (2,4)
           tick2                 -- (4,6)
```

```
runState test 0                  -- (4,6)
```

https://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-State-Lazy.html

Global Variable Example (2)

```
import Control.Monad.Trans.State

tick :: State Int Int
tick = do  n <- get              -- read Int state
           put (n+1)             -- write Int state
           return n

tick2 :: State Int Int
tick2 = do n <- get              -- read Int state
           put (n+2)             -- write Int state
           return n

test  = do tick                  -- (0,1)
           tick                  -- (1,2)
           tick2                 -- (2,4)
           tick2                 -- (4,6)
```

```
runState test 0                  -- (4,6)
```

```
test  = do n <- get
           put (n+1)
           return n

           n <- get
           put (n+1)
           return n

           n <- get
           put (n+2)
           return n

           n <- get
           put (n+2)
           return n
```

https://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-State-Lazy.html

IO Monad – return method

The **return** function takes x

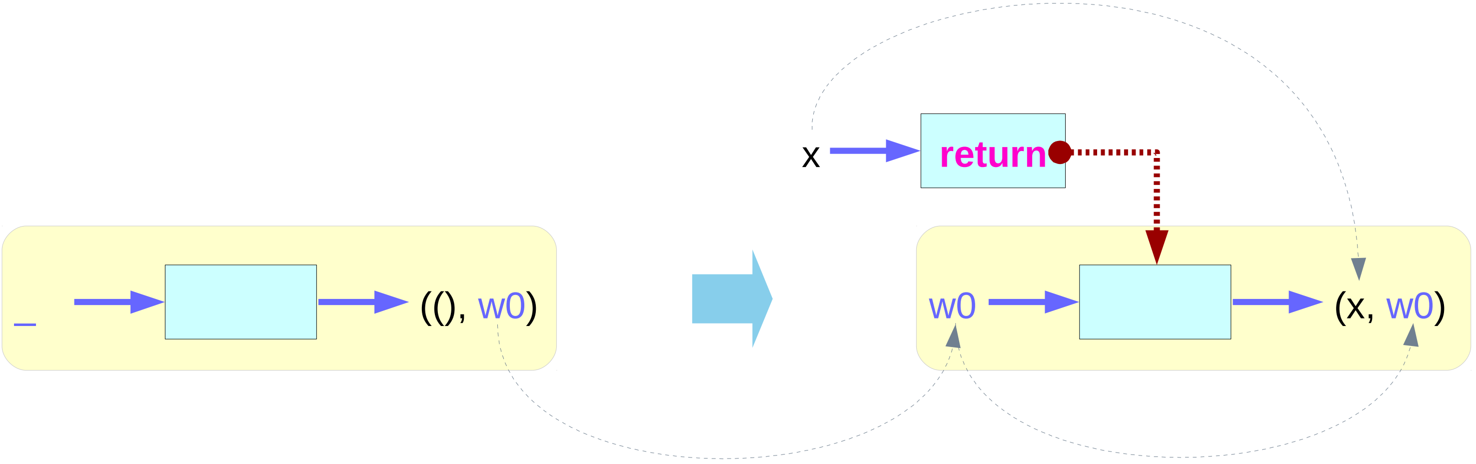
and gives back a function

that takes a w0 :: World

and returns x along with the updated World,

but not modifying the given w0 :: World

.



https://www.cs.hmc.edu/~adavidso/monads.pdf

IO Monad – actions and a result

Recall that interactive programs in Haskell are written
using the type **IO a** of **actions** that return a **result** of type **a**,
but may also perform some input/output.

A number of primitives are provided for building values of **IO a** type

```
return  :: a -> IO a
(>>=)   :: IO a -> (a -> IO b) -> IO b
getChar :: IO Char
putChar :: Char -> IO ()
```

The use of **return** and **>>=** means that **IO** is **monadic**,
and hence that the **do notation** can be used to write interactive programs.

https://www.seas.upenn.edu/~cis552/11fa/lectures/monads2.html

IO Monad – a special state monad

the **IO monad** can be viewed as a special case of the **state monad**,
in which the internal state is a suitable representation
of the state of the world:

```
type World = ...
type IO a  = World -> (a,World)
```

That is, an **action** can be viewed as a **function**
that takes the current state of the world as its argument,
and produces a value and a modified world as its result,
in which the modified world *reflects* any I/O performed by the action.

In reality, Haskell systems such as **Hugs** and **GHC** implement actions
in a more efficient manner, but for the purposes of understanding
the behavior of actions, the above interpretation can be useful.

https://www.seas.upenn.edu/~cis552/11fa/lectures/monads2.html

IO Monad – imperative procedures

IO is a type of **imperative procedures**—
actions that can have **side-effects** when executed.
A value of **IO Int**, for example, is
a **procedure** that can do **input** and **output**
and, when it's done, returns a **value** of type **Int**.

The most basic examples include
reading and writing from **STDIN** and **STDOUT**:

```
readLn :: Read a => IO a
putStrLn :: String -> IO ()
```

readLn is a procedure that
consumes a line of input from **STDIN**
and parses it with the **read** function

putStrLn is a function that,
given a **string**,
returns a procedure that prints
that string to **STDOUT**
followed by a newline.

https://www.quora.com/What-is-an-IO-Monad

IO Monad – independent execution and evaluation

The **IO procedures** that we produce are **executed**
by Haskell's **runtime system** which takes care of calling
the appropriate *OS syscalls* and *libraries* for actual effects,
as well as providing infrastructure like a *lightweight thread scheduler*.

this **execution** step is orthogonal to **evaluation.**
this simplifies normal Haskell function implementations

It is possible to **evaluate** an **IO action** without **executing** it
(using **seq**, for example), and the semantics of
how the **effects** of an **IO action** are **executed**
do not depend on how that **IO action** was **evaluated**.

https://www.quora.com/What-is-an-IO-Monad

IO **Monad** – executing IO monadic value

The **IO procedures** are to be **<u>executed</u>**

IO monadic values <u>differs</u> from

expressions in an **imperative language**

we produce these **expressions** (**IO monadic values**)

just like we produce any other sort of **value**,

evaluation

then the **expressions** are **executed** by a <u>separate</u> **interpreter**

with its own <u>semantics</u> and <u>behavior</u> (ie the **runtime** system).

execution

https://www.quora.com/What-is-an-IO-Monad

seq

A common misconception regarding **seq** is that **seq** x "evaluates" x.
seq doesn't evaluate anything just by virtue of existing in the source file,
all it does is introduce an artificial **data dependency**
when the **result** of **seq** is **evaluated**,
the first **argument** must also be evaluated.

suppose **x :: Integer**, then **seq x b** behaves essentially like
if x == 0 then b else b – unconditionally equal to **b**,
but forcing **x** along the way.

the expression **x `seq` x** is completely redundant,
and always has exactly the same effect as just writing x.

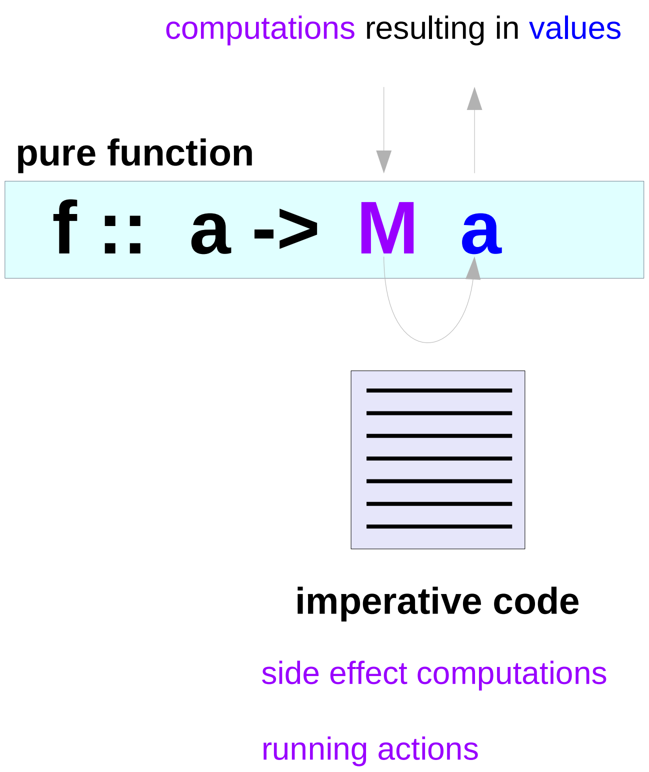
https://www.quora.com/What-is-an-IO-Monad

Pure functions and computations

Haskell separates **pure functions** from **computations**
where **side effects** must be considered

by encoding those **side effects** as **values** of a particular type.
Specifically, a **value** of type (**IO a**) is an **action**,
which if executed would produce a **value** of type **a**.

f :: Int -> World -> (Int, World)	**non-pure (side effects)**
IO a = World -> (a, World)	**pure**
f :: Int -> IO Int	



https://wiki.haskell.org/Introduction_to_IO

The only running IO action

getLine :: IO String

putStrLn :: String -> IO ()

randomRIO :: (Random a) => (a,a) -> IO a

Ordinary Haskell **evaluation** does not cause this **execution** to occur.

A **value** of type (**IO a**) is almost completely inert.

In fact, **the only IO action** which can really be said to **run**
in a compiled Haskell program is **main**.

https://wiki.haskell.org/Introduction_to_IO

x >> y

```
main :: IO ()
main = putStrLn "Hello, World!"
```

composing and chaining together IO actions

```
(>>) :: IO a -> IO b -> IO b
```

if **x** and **y** are **IO actions**, then (**x >> y**) is the action that performs **x**, dropping the result, then performs **y** and returns its result.

```
main = putStrLn "Hello" >> putStrLn "World"
```

https://wiki.haskell.org/Introduction_to_IO

x >>= y

x >>= f is the **action** that first performs the action **x**,
and captures its result, passing it to **f**,
which then computes a second action to be performed.
That action is then carried out, and its result is
the result of the overall computation.

```
main = putStrLn "Hello, what is your name?"
     >> getLine
     >>= \name -> putStrLn ("Hello, " ++ name ++ "!")
```

https://wiki.haskell.org/Introduction_to_IO

return

turns a **value** into an **IO action**

which does nothing, and simply returns that value.

at the end of a chain of actions,

we may want to decide what to return ourselves,

rather than leaving it up to the last action in the chain.

return :: a -> IO a

https://wiki.haskell.org/Introduction_to_IO

do-notation, **v <- x**

```
main = do
    putStrLn "Hello, what is your name?"
    name <- getLine
    putStrLn ("Hello, " ++ name ++ "!")

main = putStrLn "Hello, what is your name?"
   >> getLine
   >>= \name -> putStrLn ("Hello, " ++ name ++ "!")
```

An **action** on its own on a line in a do-block will be **executed**,

v <- x will cause the **action x** to be run,
and the **result** bound to the **variable v**.

https://wiki.haskell.org/Introduction_to_IO

Variable binding **v <- x**

A common mistake is to put something other than an **action** in the place of **x**, usually some other value.

If you want to make a **variable binding** inside a **do**-block which doesn't involve running an **action**, then you can use a line of the form **let a = b**, which, like an ordinary **let**-expression will define **a** to be the same as **b**, but the definition **scopes** over the remainder of the **do**-block.

v <- x

action x (monadic value)

let a = b

non-action b

https://wiki.haskell.org/Introduction_to_IO

Unsafe function

Note that there is no function:

unsafe :: IO a -> a

as this would defeat the **referential transparency** of Haskell --
applying **unsafe** to the same **IO action** might return
different **values** every time (not allowed in Haskell)

Most **monads** are actually rather **unlike IO**,
but they do share the similar concepts of bind and return.

https://wiki.haskell.org/Introduction_to_IO

Extracting **Int** from **IO Int**

```
do
   x <- returningIO
   returningIO2 $ pureFunction x
```

no way to get the "**Int**" out of an "**IO Int**",

except to do something else in the **IO Monad**.

In monad terms, the above code desugars into

```
returningIO >>= (\x -> returningIO2 $ pureFunction x)
```

https://stackoverflow.com/questions/4235348/converting-io-int-to-int

No escape from a monad

```
returningIO >>= (\x -> returningIO2 $ pureFunction x)
```

The **>>=** operator (pronounced "**bind**")
does convert the "**IO Int**" into an "**Int**",
but it does not give that **Int** directly.

It will only pass that value to a **function** as an **argument**,
and that **function** must return **another monadic value in "IO"**.

```
>>= :: IO a -> (a -> IO b) -> IO b
```

you can process the Int, but the results of doing so
never escape from the **IO monad**.

https://stackoverflow.com/questions/4235348/converting-io-int-to-int

Adding monad values

```
addM :: (Monad m, Num a) => m a -> m a -> m a
addM ma mb = do
    a <- ma
    b <- mb
    return (a + b)
```

```
addM ma mb =
    ma >>= \a ->
    mb >>= \b ->
    return (a + b)
```

```
addM ma mb =
    ma >>= (\a -> mb >>= (\b -> return (a + b)))
```

https://stackoverflow.com/questions/7840126/why-monads-how-does-it-resolve-side-effects

Using **Identity** monad instance

```
instance Monad Identity where
   return a = Identity a            -- create an Identity value
   (Identity a) >>= f = f a         -- apply f to a

addM (Identity 1) (Identity 2)

(Identity 1) >>= (\a -> (Identity 2) >>= (\b -> return (a + b)))
(\a -> (Identity 2) >>= (\b -> return (a + b)) 1
(Identity 2) >>= (\b -> return (1 + b))
(\b -> return (1 + b)) 2
return (1 + 2)
Identity 3
```

https://stackoverflow.com/questions/7840126/why-monads-how-does-it-resolve-side-effects

Using **List** monad instance

```
addM [1, 2] [3, 4]
[4,5,5,6]

addM [1, 2] [3, 4, 5]
[4,5,6,5,6,7]
```

https://stackoverflow.com/questions/7840126/why-monads-how-does-it-resolve-side-effects

Using **Maybe** instance

```
addM (Just 1) (Just 2)
Just 3

addM Nothing (Just 1)
Nothing >>= (\a -> (Just 1) >>= (\b -> return (a + b)))      – immediately abort
Nothing

addM (Just 1) Nothing:
(Just 1) >>= (\a -> Nothing >>= (\b -> return (a + b)))
(\a -> Nothing >>= (\b -> return (a + b)) 1
Nothing >>= (\b -> return (1 + b))      – immediately abort
Nothing
```

https://stackoverflow.com/questions/7840126/why-monads-how-does-it-resolve-side-effects

Using **IO** instance

```
addM (return 1 :: IO Int) (return 2 :: IO Int)
3

f :: IO Int

add :: Num a => a -> a -> a
add a b = a + b                   … side effect free

add a b = a + b + f               … side efffect monad
                                      but compile error
```

Prelude Control.Monad.Trans.State> **3 + 4 + (return 5 :: IO Int)**

<interactive>:36:1: error:

- No instance for (Num (IO Int)) arising from a use of ‘+’
- In the expression: 3 + 4 + (return 5 :: IO Int)
  In an equation for ‘it’: it = 3 + 4 + (return 5 :: IO Int)

https://stackoverflow.com/questions/7840126/why-monads-how-does-it-resolve-side-effects

Using **IO** instance

```
f :: IO Int
readLn : Read a => IO a
readLn : Read Int => IO Int

add a b =  do
   c <- readLn
   print (a + b + c)                -- not (a + b +c)

add 10 20
5
35
```

https://stackoverflow.com/questions/7840126/why-monads-how-does-it-resolve-side-effects

Using **IO** instance

in the case of **IO**, **ST** and friends,

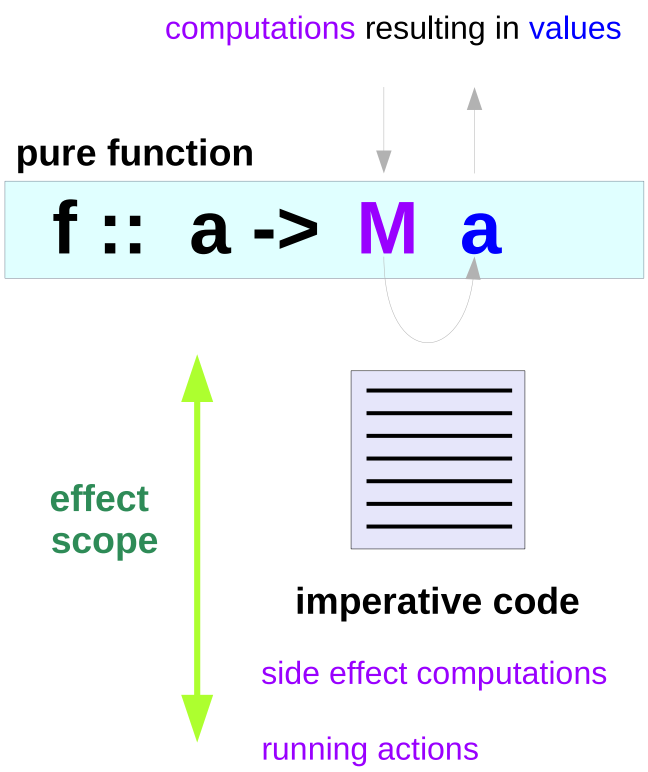
the type system keeps **effects** isolated to some specific **context**.

It does not eliminate **side effects**,

making code **referentially transparent** that should not be,

but it does determine at compile time

what **scope** the **effects** are limited to.



https://stackoverflow.com/questions/7840126/why-monads-how-does-it-resolve-side-effects

Chaining

```
f2 :: IO ()
f2 = do
    a <- f
    print a
    b <- f
    print b
```

a handy way of expressing a sequence of effects:

https://stackoverflow.com/questions/7840126/why-monads-how-does-it-resolve-side-effects

Policies for chaining computations

A monad represents some policy for **chaining** computations.

Identity's policy is **pure function composition**,

Maybe's policy is function composition with **failure propogation**,

IO's policy is **impure function composition** and so on.

https://stackoverflow.com/questions/7840126/why-monads-how-does-it-resolve-side-effects

Monad Definition

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

m a:
- Maybe a
- IO a
- ST a
- State s a

1) return

2) bind (>>=)

3) then (>>)

4) fail

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

Either Monad

a **do** block looks very much like **imperative code**
with hidden side effects.

The **Either** monadic code looks like
using **functions** that can throw **exceptions**.

data Either a b

used to represent a value which is *either* correct *or* an error;
the **Left constructor** is used to hold an **error value**
and the **Right constructor** is used to hold a **correct value**

data Either error_constructor correct_constructor

https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/12-State-Monad

Either Monad Constructors

```
data Either a b

the Left constructor : an error value
the Right constructor : a correct value

    let s = Left "foo" :: Either String Int
        s       ⇒   Left "foo"             -- error value

    let n = Right 3 :: Either String Int
        n       ⇒   Right 3                -- correct value

:type s         ⇒   s :: Either String Int
:type n         ⇒   n :: Either String Int
```

http://hackage.haskell.org/package/base-4.12.0.0/docs/Data-Either.html

Either Monad and **fmap**

The **fmap** will ignore **Left** values,
but will apply the supplied function
to values contained in a **Right**:

let s = Left "foo" :: Either String Int
let n = Right 3 :: Either String Int

fmap (*2) s	➡	**Left "foo"**	-- error value
fmap (*2) n	➡	**Right 6**	-- applied value

http://hackage.haskell.org/package/base-4.12.0.0/docs/Data-Either.html

Either Monad Example (1)

different error messages for different errors :

use **Either** to represent computations which might return

either an **error message** or a **value**:

```
myDiv3 :: Float -> Float -> Either String Float
myDiv3 x 0 = Left "Divison by zero"
myDiv3 x y = Right (x / y)

example3 x y =
  case myDiv3 x y of
    Left msg -> putStrLn msg              -- error value
    Right q    -> putStrLn (show q)       -- correct value
```

http://www.randomhacks.net/2007/03/10/haskell-8-ways-to-report-errors/

Either Monad Example (2)

can combine computations

```
divSum3 :: Float -> Float -> Float -> Either String Float
divSum3 x y z = do
    xdy <- myDiv3 x y
    xdz <- myDiv3 x z
    return (xdy + xdz)
```

used to recover from multiple kinds of **non-IO errors**

division by zero

http://www.randomhacks.net/2007/03/10/haskell-8-ways-to-report-errors/

References (1)

[1] http://www.idryman.org/blog/2014/01/23/yet-another-monad-tutorial/

[2] https://stackoverflow.com/questions/16892570/what-is-in-haskell-exactly

[3] https://wiki.haskell.org/Polymorphism

[4] https://stackoverflow.com/questions/52058692/the-term-function-application-in-haskell

[5] https://ghc.haskell.org/trac/ghc/wiki/TypeApplication

[6] https://stackoverflow.com/questions/16892570/what-is-in-haskell-exactly

[7] https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/12-State-Monad

[8] https://wiki.haskell.org/Global_variables

[9] http://www.idryman.org/blog/2014/01/23/yet-another-monad-tutorial/

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

[11] https://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-State-Lazy.html

[12] https://www.cs.hmc.edu/~adavidso/monads.pdf

[13] https://www.seas.upenn.edu/~cis552/11fa/lectures/monads2.html

[14] https://www.quora.com/What-is-an-IO-Monad

References (2)

[15] https://wiki.haskell.org/Introduction_to_IO

[16] https://stackoverflow.com/questions/4235348/converting-io-int-to-int

[17] https://stackoverflow.com/questions/7840126/why-monads-how-does-it-resolve-side-effects

[18] https://en.wikibooks.org/wiki/Haskell/Understanding_monads

[19] http://hackage.haskell.org/package/base-4.12.0.0/docs/Data-Either.html

[20] http://www.randomhacks.net/2007/03/10/haskell-8-ways-to-report-errors/

References

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

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