Monad P2 : State Transformer Monads (1C)

Young Won Lim
10/22/19

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

1. **State** Monad Control.Monad.State.Lazy
2. **IO** Monad System.IO
3. **ST** Monad Control.Monad.ST

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

State Monad

Control.Monad.State.Lazy

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

State Monad


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

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)
```

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

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) }**

single-field record

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

a function wrapped in a record syntax

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

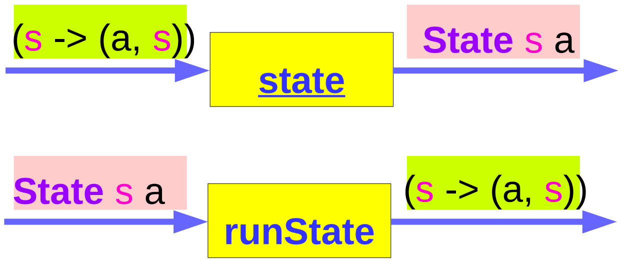
```
newtype State s a = State { runState :: s -> (a, s) }
```

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

a library function

- the accessor function **runState** is provided

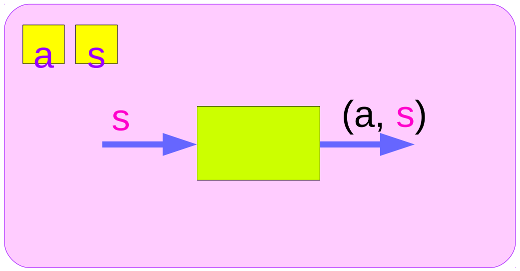
Control.Monad.Trans.State
exports a **state** function



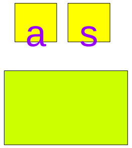
https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

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



binding variable type

state processor

run State Processor (Function)

The "state" function

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

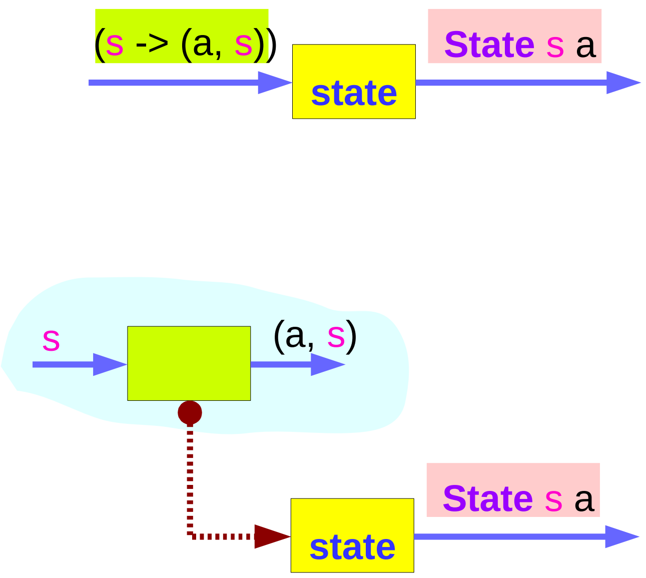
no State data constructor

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))

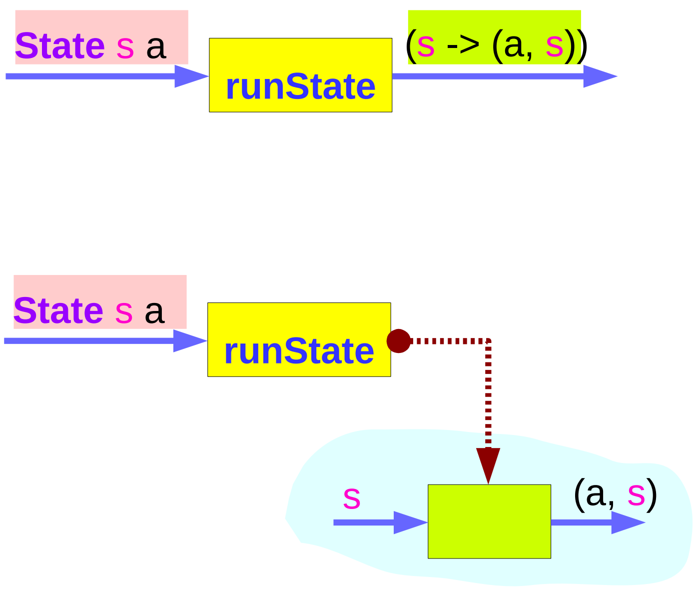
runState converts a value of type State s a

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)**



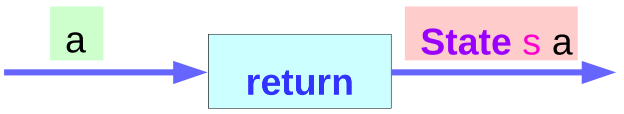

```
newtype State s a = State { runState :: s -> (a, s) }
```

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

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**

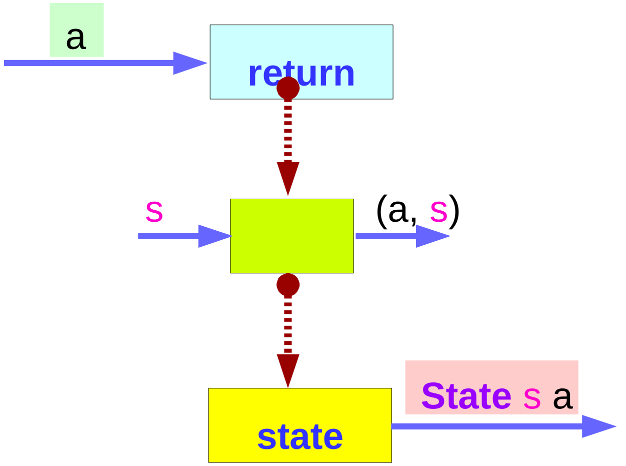
results in a **state processor** function

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

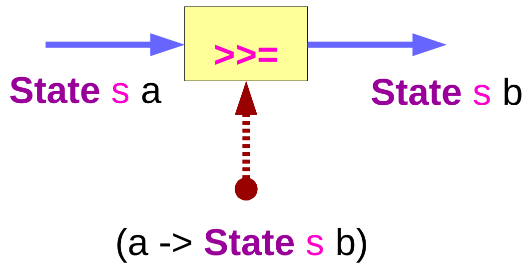
Function type of >>=

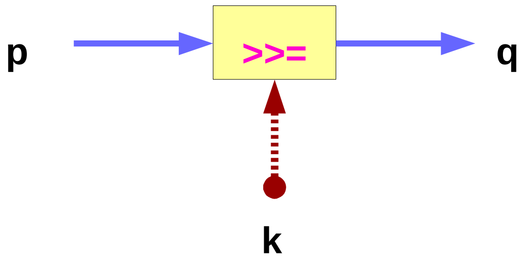
```
instance Monad (State s) where

(>>=) :: 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

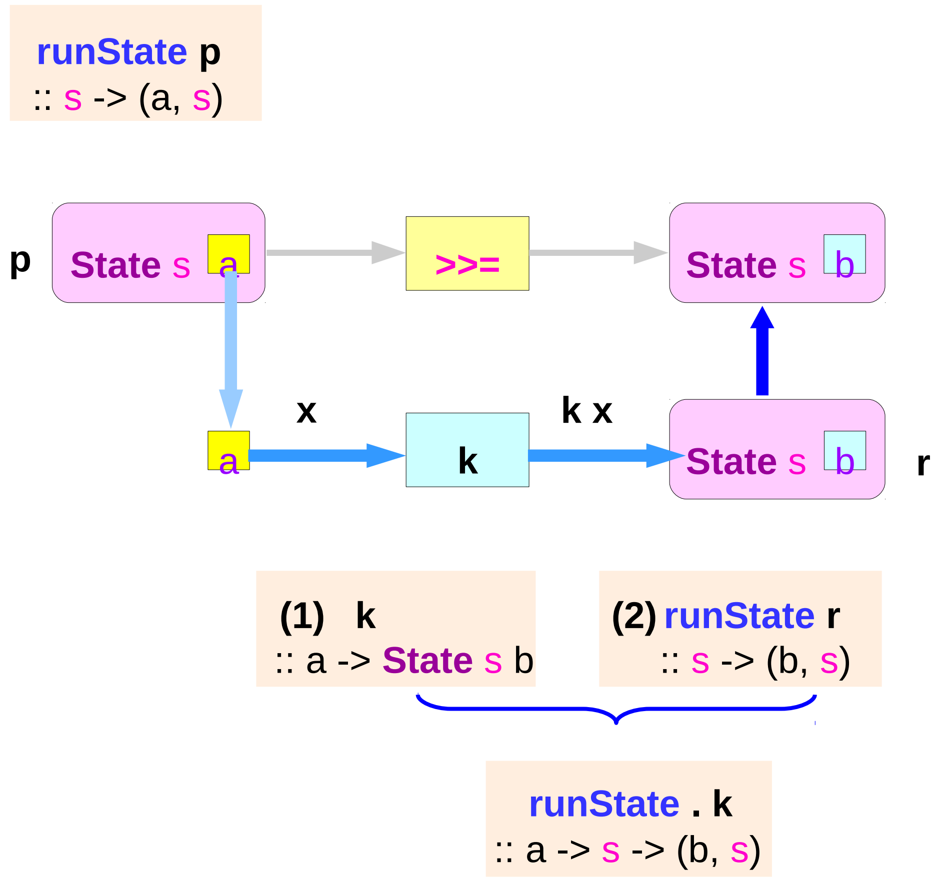




https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

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

IO Monad

System.IO

http://hackage.haskell.org/package/base-4.12.0.0/docs/System-IO.html

IO Monad Value

A **value** of **type IO a** is a **computation** which,
when performed, does some **I/O actions**
before returning a **value** of **type a**.

IO is a **monad**, so **IO actions** can be combined
using either the **do**-notation or
the **>>** and **>>=** operations
from the **Monad class**.

http://hackage.haskell.org/package/base-4.12.0.0/docs/System-IO.html#g:1

An IO action and a function

an **action** can be viewed as a **function**

- takes the **current state** of the **world** as its argument,
- produces a **value** and a **modified world** as its result,

the **modified world** reflects

any input/output performed by the action.

In reality, Haskell systems **Hugs** and **GHC**

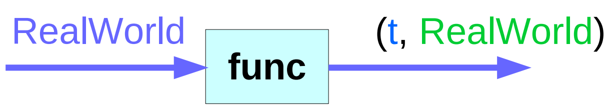
implement **actions** in a more efficient manner,

but for the purposes of understanding

the behaviour of **actions**,

the above interpretation can be useful.

func :: RealWorld -> (a, RealWorld)



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

Performing IO actions

There is really only one way to perform an **I/O action**:

bind an **I/O action** to **Main.main** in your program.

main = …

when your program is run, the **I/O** will be performed.

It is not possible to perform **I/O** from an arbitrary function,

unless that function is itself in the **IO monad**

and called at some point, directly or indirectly,

from **Main.main**.

Thread

http://hackage.haskell.org/package/base-4.12.0.0/docs/System-IO.html#g:1

Methods returing an IO monad value

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 this type, including:

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

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

Do notation in interactive programs

The use of **return** and **>>=** means that **IO** is **monadic**,
and hence that the **do notation** can be used
to write **interactive programs**.
For example, the action that reads a string of characters
from the keyboard can be defined as follows:

```
getLine :: IO String
getLine =  do x <- getChar
              if x == '\n' then
                  return []
              else
                  do    xs <- getLine
                        return (x:xs)
```

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

Sequencing IO Operations – using **>>=** and **>>**


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

main =     readFile "in-file"                          >>= \s ->
           writeFile "out-file" (filter isAscii s)     >>
           putStr "Filtering successful\n"
```

https://www.haskell.org/onlinereport/haskell2010/haskellch7.html

Sequencing IO Operations – using **do** notation

```
main = do
        putStr "Input file: "
        ifile <- getLine
        putStr "Output file: "
        ofile <- getLine
        s <- readFile ifile
        writeFile ofile (filter isAscii s)
        putStr "Filtering successful\n"
```

https://www.haskell.org/onlinereport/haskell2010/haskellch7.html

IO Monad Definition Summary


```
newtype IO a = IO (State# RealWorld -> (# State# RealWorld, a #))

instance  Monad IO  where
    return    = returnIO
    (>>=)     = bindIO

returnIO :: a -> IO a
returnIO x = IO $ \s -> (# s, x #)

bindIO :: IO a -> (a -> IO b) -> IO b
bindIO (IO m) k = IO $ \s -> case m s of (# new_s, a #) -> unIO (k a) new_s
```

GHC.Types

System.IO

http://blog.ezyang.com/2011/05/unraveling-the-mystery-of-the-io-monad/

IO Monad Type

The **IO type** is just a **newtype** defined in **GHC.Prim / GHC.Types**:

newtype IO a = IO (State# RealWorld -> (# State# RealWorld, a #))

GHC.Types

Look at a naive implementation of **State monad**:

newtype State s a = State (s -> (s, a))

State# RealWorld ⟷ **s**

https://stackoverflow.com/questions/19093016/why-cant-i-use-io-constructor/19093720

IO Monad Type

```
newtype IO a = IO (State# RealWorld -> (# State# RealWorld, a #))
```

the argument of IO **constructor**
is not the same as the argument of **return**.

```
….
return    = returnIO
..

returnIO :: a -> IO a
returnIO x = IO $ \ s -> (# s, x #)
```

GHC.Types

State# RealWorld ⟷ **s**

https://stackoverflow.com/questions/19093016/why-cant-i-use-io-constructor/19093720

Abstract IO Monad Type

IO is an **abstract type**:

it's an intentional decision not to export the **constructor** (**IO**)
so you can neither construct **IO** nor pattern match it.

This allows Haskell to enforce **referential transparency**
and other useful properties even in presence of input-output.

https://stackoverflow.com/questions/19093016/why-cant-i-use-io-constructor/19093720

Abstract and strict type RealWorld

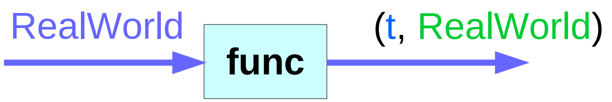
The **RealWorld** type is an **abstract** datatype,
so **pure functions** also can't construct
RealWorld values by themselves,

The **RealWorld** type is a **strict** type,
so **undefined** also can't be used.

newtype IO a = IO (State# RealWorld -> (# State# RealWorld, a #))

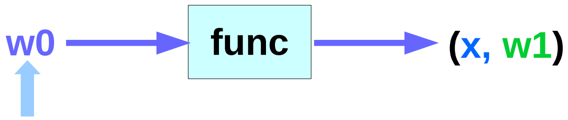
State# RealWorld	(abstract type)
IO a	(abstract type)

func :: RealWorld -> (a, RealWorld)



Executing an IO action

func w0 = (x, w1)



https://wiki.haskell.org/IO_inside#IO_actions_as_values

Abstract data types

s **type** with associated **operations**,
but whose representation is **hidden**.

Abstract data type examples:

- the built-in **primitive types**, **Integer** and **Float**.
- **parametrized types** : as a kind of abstract type,
  because some parts of the data type is
  undefined, or **abstract**.

the **interface** is the **set** of **operations**
that can be used to manipulate **values** of the data type.

does not manipulate the **part** of the data type that was left **abstract**.

https://wiki.haskell.org/IO_inside#IO_actions_as_values

A special case of **State** monad

it is interesting to note that 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)
```

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

Generic definition

```
instance Monad IO where
    return x w0 = (x, w0)

    (ioX >>= f) w0  =
        let    (x, w1) = ioX w0
        in     f x w1                -- has type (t, World)
```

```
type   IO t   =   World   ->   (t, World)
```

type synonym

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

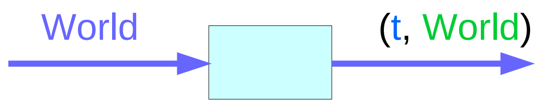
Type Synonym **IO t**

IO t is a **parameterized <u>function</u> type**

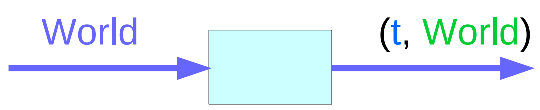
input : a World

output: a result value of the type t and a new updated World
are obtained by modifying the given World
in the process of computing the result value of the type t.

World -> (t, World)



IO t



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

type synonym

cf) type application

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

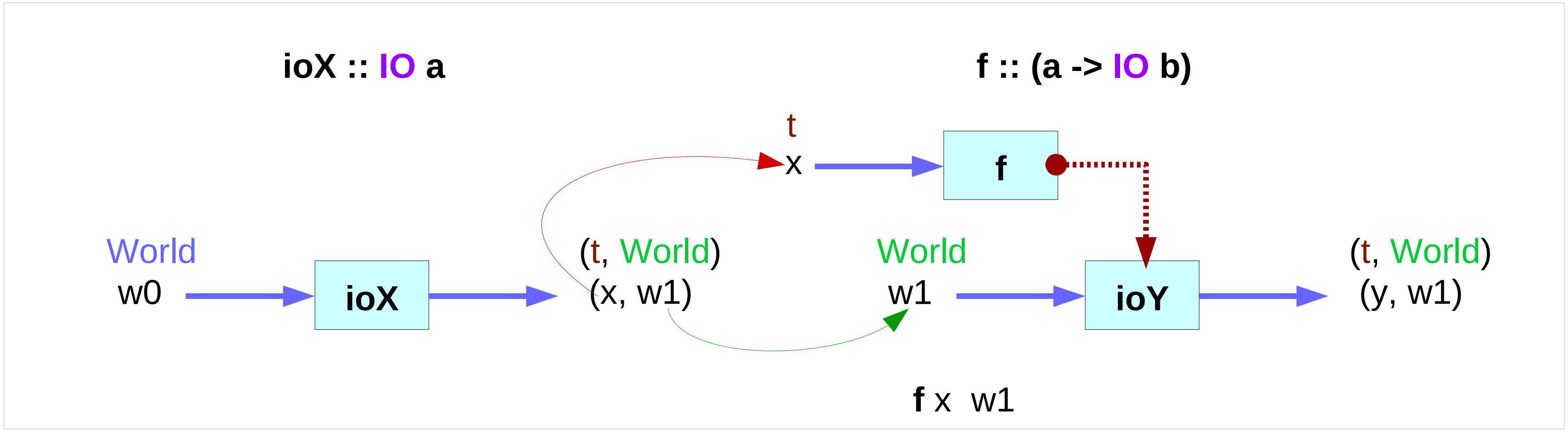
(>>=) bind operator explained

```
instance Monad IO where
    return x world = (x, world)
    (ioX >>= f) world0  =
    let
        (x, world1) = ioX world0
    in
        f x world1                          -- Has type (t, World)
```

(ioX >>= f) :: IO a -> (a -> IO b) -> IO b

ioX :: IO a

f :: (a -> IO b)



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

return method

The return function takes **x**

and gives back a **function**

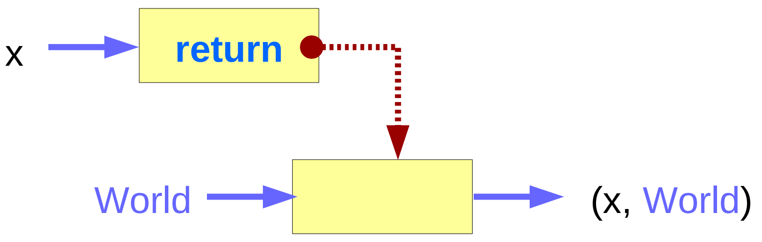
that takes a **World**

and returns **x** along with the new, updated **World**

formed by not modifying the **World** it was given

.

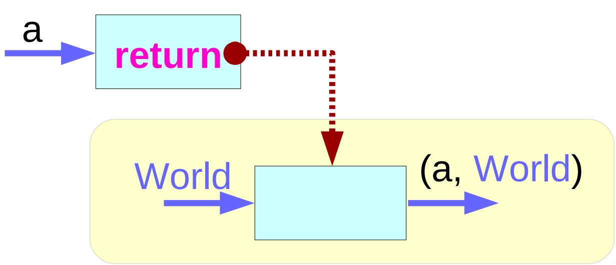
return x world = (x, world)



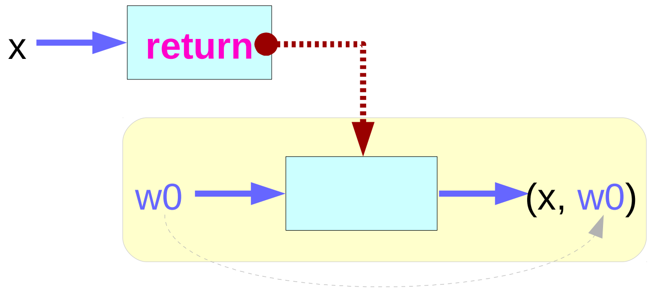
https://www.cs.hmc.edu/~adavidso/monads.pdf

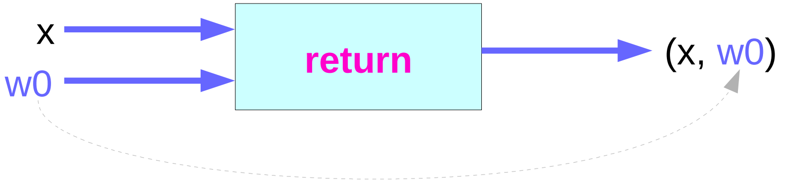
return method and partial application

return a :: a -> IO a ⟸ **Types** ⟹ **return a World :: (a, World)**



let (x, w0) = return x w0 ⟸ **Values** ⟹ **let (x, w0) = return x w0**





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

bind method **(>>=)**

the expression (**ioX >>= f**) has type **World -> (t, World)**

a function that takes a World, called w0,

which is used to extract x from its **IO** monad.

This gets passed to **f**, resulting in another **IO** monad,

which again is a function that takes a World

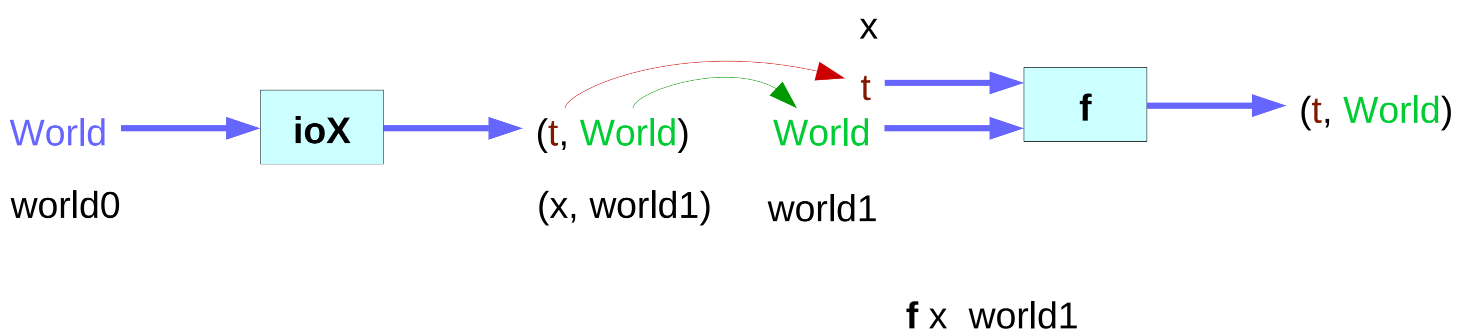
and returns a x and a new, updated World.

We give it the World we got back from getting x out of its monad,

and the thing it gives back to us is the t with a final version of the World

.

the implementation of bind



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

ST Monad

Control.Monad.ST

http://hackage.haskell.org/package/base-4.12.0.0/docs/Control-Monad-ST.html

ST, **IO**, and **State** monads

ST monad

- a more *powerful version* of the **State** monad
- was *originally* written
  to provide Haskell with **IO** capability

IO monad is basically just

a **State** monad with an **environment**

of all the information about the **real world**.

inside GHC at least, **ST** is used,

and the **environment** is a **type** called **RealWorld**.

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

ST Monad – mutable state

data ST s a

an **ST computation**

- an **internal** **state** is used to produce **results**
  (**ST s a** is similar to **State s a**)

- the **state** is **mutable**
  (**ST s a** is different from **State s a**)

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

ST Monad – mutable state

data ST s a

an **ST computation**

- an **internal** **state** is used to produce **results**
  (**ST s a** is similar to **State s a**)

- the **state** is **mutable** ……… **mutable variable**
  (**ST s a** is different from **State s a**)

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

ST Monad – imperative code enabled

functions written using the **ST** **monad**

appear completely **pure** to the rest of the program.

This allows programmers to produce **imperative code**

where it may be impractical to write **functional code**,

while still keeping all the **safety** that **pure code** provides.

https://en.wikipedia.org/wiki/Haskell_features#ST_monad

Pure functional language

In a **pure** functional language,

you can't do anything that has a **side effect**.

A **side effect** would mean that **evaluating** an expression

changes some **internal state** that would later cause

evaluating the same expression to have a **different result**.

https://stackoverflow.com/questions/4382223/what-does-pure-mean-in-pure-functional-language

Side effect example

For example, a pure functional language cannot

- have an **assignment operator** …. (imperative code)
- or do **input/output** ……………….. (IO monad)

although for practical purposes,

even pure functional languages

often call impure libraries to do I/O.

https://stackoverflow.com/questions/4382223/what-does-pure-mean-in-pure-functional-language

ST monad advantage

The **ST** **monad** allows programmers

to write **imperative algorithms** in Haskell,

by using mutable **variables** (**STRef**'s)

and mutable **arrays** (**STArrays** and **STUArrays**).

- **code** can have internal **side effects**
  - destructively updating
    mutable **variables** and **arrays**,
  - containing these **effects** inside the monad.

https://en.wikipedia.org/wiki/Haskell_features#ST_monad

Imperative coding style using **STRef** Monad

a version of the function sum is defined,
in a way that **imperative languages** are used

a **variable** is directly updated, ……………………….. (imperative style)
rather than a **new value** is formed and …………….. (functional style)
passed to the **next iteration** of the function.

While in place modifications of the **n :: STRef s a** are occurring,
something that would usually be considered a **side effect**,
it is all done in a safe way which is deterministic.

Memory modification in place is possible
While maintaining the **purity** of a function by using **runST**

https://wiki.haskell.org/Monad/ST

ST Monad

```
data ST s a
```

```
newtype ST s a = ST (State# s -> (# State# s, a #))
```

```
newtype ST s a = ST (STRep s a)
type STRep s a = State# s -> (# State# s, a #)
```

ST s a looks a lot like **State s a**

An **ST computation** is one that

uses an **internal state** to produce results,

except that the **state** is **mutable**.

For mutable state,

Data.STRef provides **STRefs**.

A **STRep s a** is exactly like

an **IORep s a** ,

but it lives in the **ST s monad**

rather than in **IO**.

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

ST Monad

There is one major difference
that sets apart **ST**
from both **State** and **IO**.

Control.Monad.ST offers a **runST** function with the following type:

```
runST :: (forall s. ST s a) -> a
```

If **ST** involves **mutability**,
how come we can simply extract
a values from the monad?

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

ST Monad

The type signature.

runST :: (forall s. ST s a) -> a

The answer lies in the **forall s.** part of the type.

Having a **forall s.** enclosed within the type of an argument

amounts to telling the type checker "s could be anything.

Don't make any assumptions about it".

Not making any assumptions, however, means

that s cannot be matched with anything else –

even with the s from another invocation of **runST**

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

(ST s) Monad

```
instance Monad (ST s) where
    {-# INLINE (>>=)  #-}
    (>>) = (*>)
    (ST m) >>= k
      = ST (\s ->
        case (m s) of
          { (# new_s, r #) ->  case (k r) of
                                  { ST k2 -> (k2 new_s) }    } )

newtype ST s a = ST (STRep s a)
type STRep s a = State# s -> (# State# s, a #)
```

http://hackage.haskell.org/package/base-4.12.0.0/docs/Control-Monad-ST.html

Using a Generic State Transformer (6)

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