Monad P1 : IO Actions (5A)

Young Won Lim
6/6/19

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

This document was produced by using LibreOffice/OpenOffice.

Based on

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

IO Monad

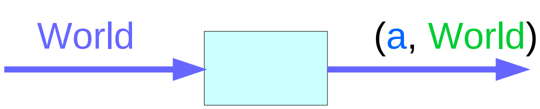
Haskell separates **pure functions** from **computations**
where **side effects** must be considered
by encoding those **side effects**
as **values** of a particular type (**IO a**)

Specifically, a **value** of type (**IO a**) is an **action**,
which *if executed* would produce a **result value** of type **a**.

Execution ⇨ Value (result)

IO a

a type of an action



https://wiki.haskell.org/Introduction_to_IO

Computations that result in values

Monads like **IO**

map types **t** to a new type **IO t**

that represent "computations that result in values"

a **function** type: **World -> (t, World)**

the **result** type : **t**

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

RealWorld -> (a, RealWorld)

https://wiki.haskell.org/Maybe

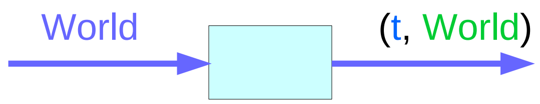
Type Synonym **IO t**

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

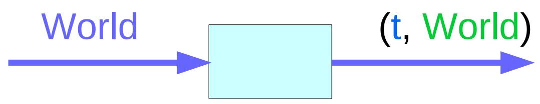
input : a World

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

World -> (t, World)



IO t



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

cf) type application

RealWorld

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

A **pure** language

the **result** of any function **call** is
fully determined by its **arguments**.

impossible to have functions like **rand()** or **getchar()** in **C**
which return different results on each **call**

can't have **side effects**
they can't effect any changes to the real world,
like changing files, writing to the screen, printing,

any **function call** can be replaced
by the **result** of a **previous call**
with the **same parameters**,

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

The problems of IO and side effects

1. repeated calls

2. the order of calls

Solution: use some artificial parameter i0, i1
incur data dependencies

```
get2chars :: Int -> (String, Int)
get2chars i0 = ([a,b], i2)  where  (a,i1) = getchar i0
                                   (b,i2) = getchar i1
```

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

main

main :: RealWorld -> ((), RealWorld)

RealWorld is a artificial parameter **type** used instead of our Int.
like the **baton** passed in a relay race.

When **main** calls some **IO function**,
it passes the **RealWorld type value** as a parameter. (**baton**)

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

IO a type synonym

main :: RealWorld -> ((), RealWorld)

type IO a = RealWorld -> (a, RealWorld)

main has type **IO ()**

getChar has type **IO Char**

think of the type **IO Char** as meaning

take the current **RealWorld**, do something to it,

and return a **Char** and a (possibly changed) **RealWorld**

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

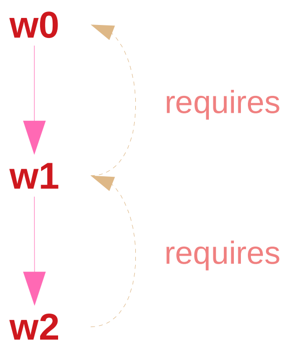
Baton values used for strict ordering

```
getChar :: RealWorld -> (Char, RealWorld)

main :: RealWorld -> ((), RealWorld)
main w0 = let   (a, w1) = getChar w0
                (b, w2) = getChar w1
            in ((), w2)
```

main calling **getChar** two times:

RealWorld values are used like a **baton** which gets passed between all routines called by '**main**' in **strict order**.

Inside each call **RealWorld** values are used in the same way.



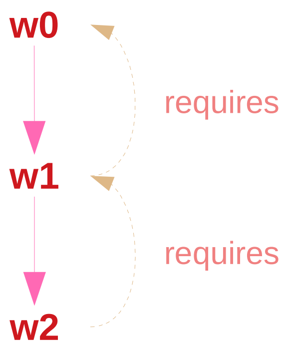
https://wiki.haskell.org/IO_inside#IO_actions_as_values

RealWorld type values

to compute the world value to be returned from **main**,
each **IO procedure** is to be performed
that is called from **main** directly or indirectly.

each **procedure** in the **chain** will be performed in sequence
just in a proper time (relative to the other **IO actions**)

cost of passing these **RealWorld** values is free!
these fake values exist only for the **compiler**
to analyze and optimize the code
but when it gets to assembly code generation,
all these parameters and result values can be removed
from the final generated code.



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

IO actions

Using **IO actions** guarantees that:

the **execution order** will be retained as written

each action will have to be **executed**

the **result** of the **same** **action** (such as "readVariable varA") will not be reused

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

Do – syntax sugar

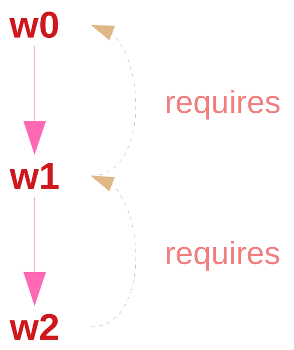
do notation eventually gets translated to
statements passing world values around and
is used to simplify the gluing of several **IO actions** together.

```
main = do putStr "What is your name?"
          putStr "How old are you?"
          putStr "Nice day!"

main = (putStr "What is your name?")
       >> ( (putStr "How old are you?")
            >> (putStr "Nice day!")
          )
```

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

Then operator (>>) – syntax sugar

```
(>>) :: IO a -> IO b -> IO b
(action1 >> action2) w0 =
   let (a, w1) = action1 w0
        (b, w2) = action2 w1
   in (b, w2)

action1 >> action2 = action
  where
    action w0 = let (a, w1) = action1 w0
                     (b, w2) = action2 w1
                in (b, w2)
```



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

Bind variable and operator (>>=)

```
main = do a <- readLn
                 print a

main = readLn
           >>= (\a -> print a)

(>>=) :: IO a -> (a -> IO b) -> IO b
(action1 >>= action2) w0 =
   let (a, w1) = action1 w0
        (b, w2) = action2 a w1
   in (b, w2)
```

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

Binding variable and operator examples

```
action1 >>= (\x -> action2)

main = do putStr "What is your name?"
          a <- readLn
          putStr "How old are you?"
          b <- readLn
          print (a,b)

main =    putStr "What is your name?"
          >>   readLn
          >>= \a -> putStr "How old are you?"
          >>   readLn
          >>= \b -> print (a,b)
```

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

return method

```
return :: a -> IO a
return a world0  =  (a, world0)

main = do a <- readLn
          return (a*2)
```

in an **imperative** language,

return immediately returns from the **IO procedure**

In Haskell, the only purpose of using **return** is

to **lift** some **value** (of type **a**)

into the **result** of a whole **action** (of type **IO a**)

used only as the last executed statement of some **IO sequence**.

```
type IO a  =  RealWorld -> (a, RealWorld)
```

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

return method examples

```
main = do a <- readLn
          when (a>=0) $ do
              return ()
          print "a is negative"
```

the 'print' statement is executed always

```
main = do a <- readLn
          if (a>=0)
            then return ()
            else print "a is negative"
```

the 'print' statement is executed only when the condition is met

```
main = do a <- readLn
     if (a>=0)
       then return ()
       else do
         print "a is negative"
         ...
```

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

Haskell layout / indentation rule

```
do first thing
second thing
third thing
```

wrong

```
do first thing
 second thing
 third thing
```

wrong

```
do first thing
   second thing
   third thing
```

OK

```
do
  first thing
  second thing
  third thing
```

OK

```
if foo
  then do first thing
       second thing
       third thing
  else do something_else
```

wrong

```
if foo
  then do first thing
          second thing
          third thing
  else do something_else
```

OK

```
if foo
  then do
    first thing
    second thing
    third thing
  else do
    something_else
```

OK

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

liftM

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

liftM f action = do x <- action
                    return (f x)
```

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

IO actions in **pure** procedures – no execution allowed

it's impossible to execute **IO actions**

inside **pure (non-IO) procedures**.

pure procedures

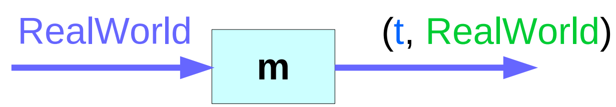
just don't get a **baton** (**w0**)

don't know any **world value** to pass to an **IO action**.

the **prohibition** of using **IO actions** inside **pure procedures**

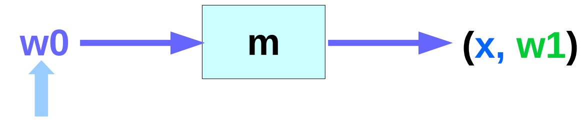
is just a type system trick (as it usually is in Haskell).

m :: RealWorld -> (a, RealWorld)



Executing an IO action

m w0 = (x, w1)



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

Abstract and strict type RealWorld

The **RealWorld** type is an **abstract** datatype,

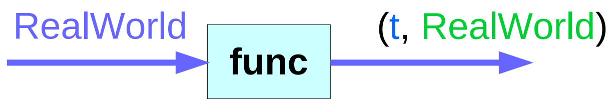
so **pure functions** also can't construct

RealWorld values by themselves,

The **RealWorld** type is a **strict** type,

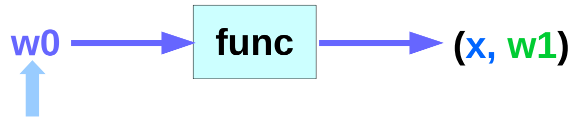
so **undefined** also can't be used.

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**.

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

Strict data types

The **strictness annotation !** on **constructor** fields
is used mainly to avoid **space leaks**

data T = T !Int !Int

neither component of the T constructor can harbour a space leak,
because both components (Int, Int) must be fully evaluated
to **Int**s when the constructor is built.

strictness annotations can make **performance** worse

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

IO actions in **pure** procedures – only as a function value

while **pure code** can't *execute* **IO actions**,

pure procedure can work with them

as with any other **functional values**

- they can be stored in data structures,
- passed as parameters,
- returned as results,
- collected in lists, and
- partially applied.

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

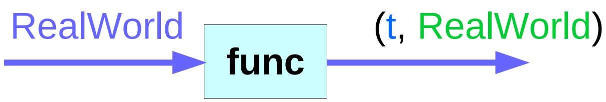
Executing IO actions in IO procedures

an **IO action** will remain just a **functional value**
in partially evaluated form, like any function
unless the **last argument** of type **RealWorld** is computed

to *execute* the **IO action** means
to compute a **value** of the type **(t, RealWorld)**

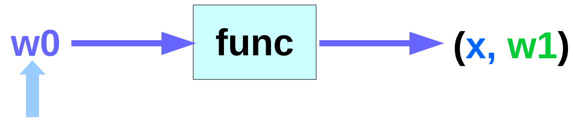
this can be done only inside some **IO procedure**,
in its **actions chain**.

func :: RealWorld -> (a, RealWorld)



Executing an IO action

func w0 = (x, w1)



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

Executing IO Actions – main chain

IO actions like **get2chars** cannot be executed directly

because they needs a **RealWorld argument**

insert a **Realworld value** in the **main** chain,

placing them in some **do** sequence executed from **main**

```
main world0 = let  get2chars = getChar >> getChar
                   ((), world1) = putStr "Press two keys" world0
                   (answer, world2) = get2chars world1
              in ((), world2)
```

either directly in the **main** function
explicit sequencing

```
main = do let get2chars = getChar >> getChar
          putStr "Press two keys"
          get2chars
          return ()
```

or indirectly in an **IO** function
Implicit sequencing

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

Executing IO actions – trigger

real **execution** of this action will take place

only when this **procedure** is called as part of the process

of calculating the final **value** of **world** for **main**.

initial value

```
main world0 = let  get2chars = getChar >> getChar
                   ((), world1) = putStr "Press two keys" world0
                   (answer, world2) = get2chars world1
               in ((), world2)
```

final value triggers

three **let** bindings
order not matter

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

Executing IO Actions – Order

```
main world0 = let  get2chars = getChar >> getChar

                   ((), world1) = putStr "Press two keys" world0

                   (answer, world2) = get2chars world1

                in ((), world2)
```

three **let** bindings

the execution order

- the **let bindings** do not constrain any **order**
- processing **world** values do constrain the **order**

arbitrary reorder the **let** binding statements

does not affect the execution order.

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

Executing IO Actions – implicit passing the world value

```
main = do let get2chars = getChar >> getChar
          putStr "Press two keys"
          get2chars
          return ()
```

do notation
sequential order

only one **let** bindings

the **non-let statements** are executed
in the exact **order** in which they're written,
they still pass the **world** value
from **statement** to **statement** as before

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

List of IO actions

```
ioActions :: [IO ()]
IoActions = [ (print "Hello!"),
              (putStr "just kidding"),
              (getChar >> return ())  ]
```

the real type of this list:

ioActions :: [RealWorld -> ((), RealWorld)]

insert them into the 'main' chain:

```
main = do head ioActions
          ioActions !! 1
          last ioActions
```

do notation
sequential order

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

List of IO actions

any **IO action** in a **do** statement or the **>>** or **>>= operators**

is an **expression** returning a result of type **IO a** for some type **a**

In a function of the type **x -> y -> ... -> IO a**

with all parameters of the types of x, y

IO a is really a function type

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

List of IO actions

a function that executes all the **IO actions** in the list:

```
sequence_ :: [IO a] -> IO ()
sequence_ [] = return ()
sequence_ (x:xs) = do x
                      sequence_ xs
```

extract IO actions from the list and
insert them into a chain of IO operations
to be executed one after another
to "compute the final world value" of the entire 'sequence_' call.

```
main = sequence_ ioActions
```

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

List methods

length xs	Get the size of the list.
reverse xs	Turn a list backwards.
xs !! n	Get the Nth element out of a list.
head xs	the first element of the list
last xs	the last element of the list
filter my_test xs	Get a list of all elements that match some condition. Returns everything that passes the test
minimum xs	the highest element of a list
maximum x	the lowest element of a list

https://wiki.haskell.org/How_to_work_on_lists

Implementation of **IO t**

It is impossible

to store the extra copies of the contents of your hard drive

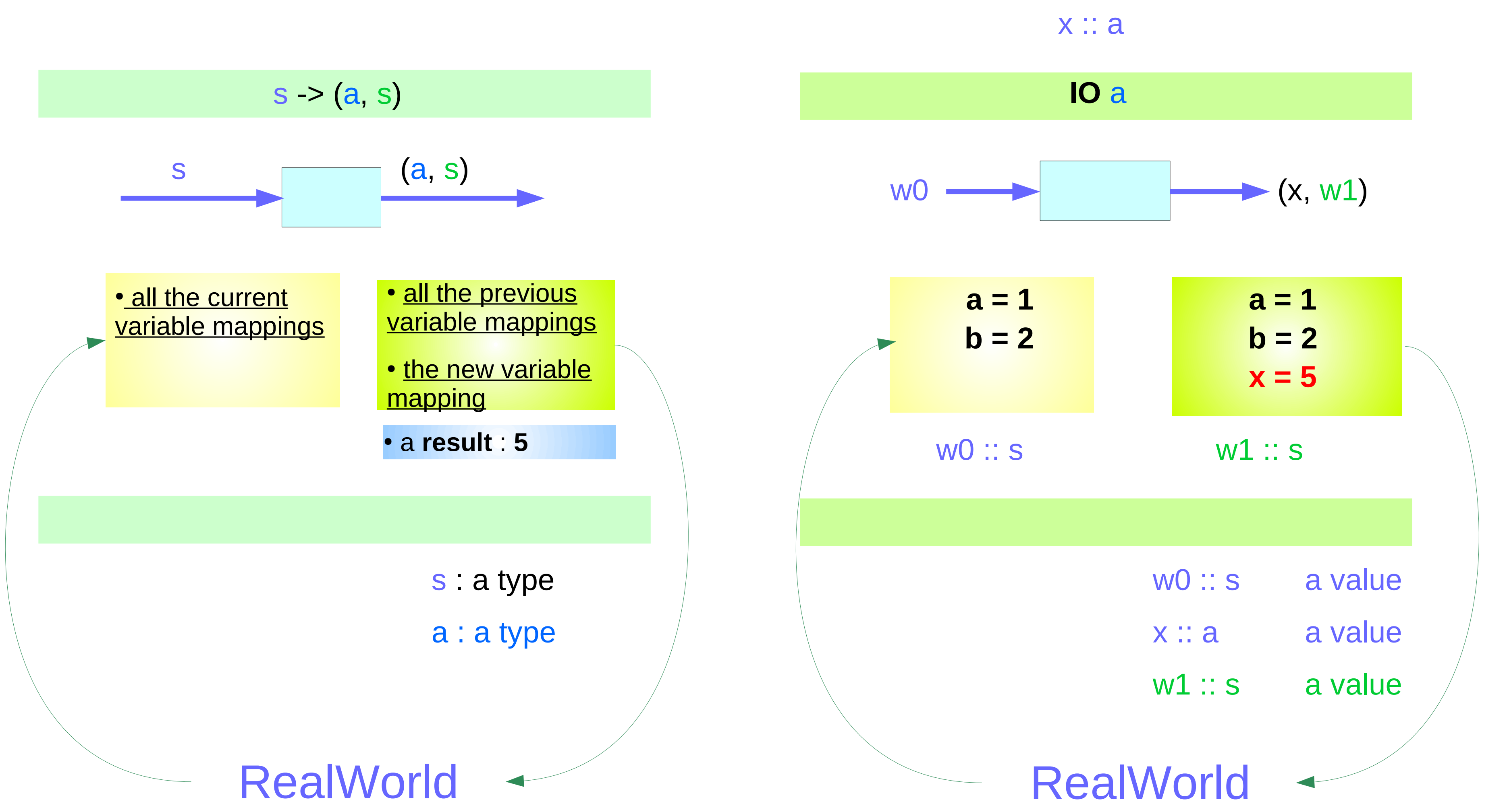
that each of the Worlds contains

given World → updated World

type IO a = RealWorld -> (a, RealWorld)

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

Variable Mappings : Context



http://learnyouahaskell.com/for-a-few-monads-more

IO Monad in GHC

Which World was given initially?

Which World was updated?

In GHC, a **main** must be defined somewhere with type **IO ()**

a program execution starts from the **main**

the initial World is contained in the **main** to start everything off

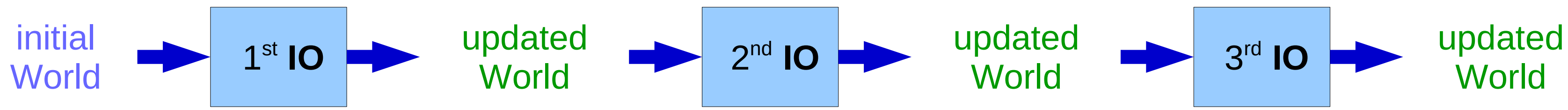
the **main** passes the updated World from each **IO**

to the next **IO** as its initial World

an **IO** that is not reachable from **main** will never be executed

an initial / updated World is not passed to such an **IO**

The modification of the World



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

IO Monad in GHCI

when using **GHCI**,

everything is wrapped in **an implicit IO**,

since the results get printed out to the screen.

there's only 1 World in existence at any given moment.

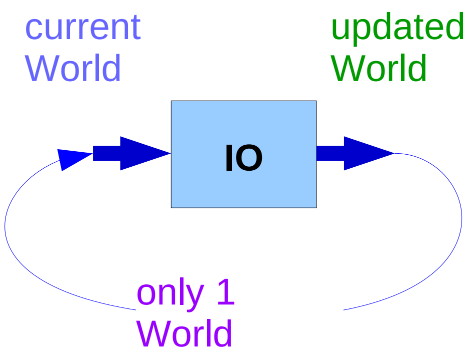
each **IO** takes that one and only World, consumes it,

and gives back a single new updated World.

consequently, there's no way to accidentally run out of Worlds,

or have multiple ones running around.

the implementation of bind



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

GHCI

Every time a new **command** is given to **GHCI**,

GHCI passes the current World to **IO**,

GHCI gets the *result* of the command back,

GHCI request to display the *result* (**executing actions**)

(which updates the World by modifying

- the contents of the screen or
- the list of defined variables or
- the list of loaded modules or whatever),

GHCI saves the new World to process the next command.

the implementation of bind

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

References

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

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