

Side Effects (3B)

Copyright (c) 2016 - 2017 Young W. Lim.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

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

This document was produced by using OpenOffice.

Based on

[Haskell in 5 steps](https://wiki.haskell.org/Haskell_in_5_steps)

https://wiki.haskell.org/Haskell_in_5_steps

Functional vs. Imperative Languages

Imperative programming:

- variables as **changeable locations** in a computer's memory
- imperative programs **explicitly commands** the computer what to do

functional programming

- a way to think in higher-level **mathematical terms**
- defining how **variables relate** to one another
- leaving the **compiler** to **translate** these
to the step-by-step instructions that the computer can process.

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

Haskell Language Features

Haskell Functional Programming

- **Immutability**
- **Recursive Definition**
- **No Data Dependency**

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

Redefinition : not allowed

imperative programming:

after setting $r = 5$ and then changing it to $r = 2$.

$r = 5$

$r = 2$

Haskell programming:

an error: "multiple declarations of r ".

Within a given scope, a **variable** in Haskell

gets defined **only once** and **cannot change**, like variables in mathematics.

$r = 5$



~~$r = 2$~~

No mutation

In Haskell

Immutable: They can change only based on the data we enter into a program.

We can't define r two ways in the same code,

but we could change the value **by changing the file**

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

Variable definition in a file

Var1.hs

```
r = 5
```

Var2.hs

```
r = 55
```

definition with initialization

```
young@Sys ~ $ ghci
GHCi, version 7.10.3: http://www.haskell.org/ghc/ :? for help
Prelude> :load Var1.hs
[1 of 1] Compiling Main          ( var.hs, interpreted )
Ok, modules loaded: Main.
*Main> r
5
*Main> :t r
r :: Integer
*Main>
*Main> :load Var2.hs
[1 of 1] Compiling Main          ( var2.hs, interpreted )
Ok, modules loaded: Main.
*Main> r
55
```

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

No Mutation

Var1.hs

r = 5

Var2.hs

r = 55

No mutation

```
*Main> r = 33
```

```
<interactive>:12:3: parse error on input '='
```

```
young@Sys ~ $ ghci
```

```
GHCi, version 7.10.3: http://www.haskell.org/ghc/ :? for help
```

```
Prelude> r = 333
```

```
<interactive>:2:3: parse error on input '='
```

```
Prelude>
```

```
let r = 33
```

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

Recursive Definition

imperative programming:

$r = r + 1$

incrementing the variable r

(updating the value in memory)

Haskell programming:

a **recursive definition** of r


(defining it in terms of itself)

if r had been defined with any value beforehand,
then $r = r + 1$ in Haskell would bring an error message.


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

No Data Dependence

$y = x * 2$
 $x = 3$

A curved arrow points from the variable 'x' in the second line to the variable 'x' in the first line, indicating that the value of 'y' depends on the value of 'x'.

$x = 3$
 $y = x * 3$

A curved arrow points from the variable 'x' in the second line to the variable 'x' in the first line, indicating that the value of 'y' depends on the value of 'x'.

Haskell programming:

because their values of variables do not change within a program

variables can be defined in any order

there is no notion of "x being declared before y" or the other way around.

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

Evaluation

area 5

=> { replace the LHS $\text{area } r = \dots$ by the RHS $\dots = \pi * r^2$ }

$\pi * 5^2$

=> { replace π by its numerical value }

$3.141592653589793 * 5^2$

=> { apply exponentiation (^) }

$3.141592653589793 * 25$

=> { apply multiplication (*) }

78.53981633974483

$\text{area } r = \pi * r^2$

replace each function with its **definition**

calculate the results until a single value remains.

to apply or call a function means

to **replace the LHS** of its **definition** by its **RHS**.

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

Side Effects Definition

a **function** or **expression** is said to have a **side effect** if it modifies some state outside its scope or has an observable interaction with its calling functions or the outside world besides returning a value.

a particular function might

- modify a global variable or static variable,
- modify one of its arguments,
- raise an exception,
- write data to a display or file,
- read data, or
- call other side-effecting functions.

[https://en.wikipedia.org/wiki/Side_effect_\(computer_science\)](https://en.wikipedia.org/wiki/Side_effect_(computer_science))

Side Effects in Haskell

In the presence of side effects,
a program's behaviour may depend on **history**;

the **order of evaluation** matters.
the **context** and **histories**

Imperative programming : frequent utilization of side effects.
functional programming : side effects are rarely used.

The lack of side effects makes it easier
to do **formal verifications** of a program

The functional language Haskell expresses
side effects such as **I/O** and
other **stateful computations**
using **monadic actions**

[https://en.wikipedia.org/wiki/Side_effect_\(computer_science\)](https://en.wikipedia.org/wiki/Side_effect_(computer_science))

Side Effects Examples in C

```
int i, j;  
i = j = 3;  
  
i = (j = 3);    // j = 3 returns 3, which then gets assigned to i
```

```
// The assignment function returns 10  
// which automatically casts to "true"  
// so the loop conditional always evaluates to true  
  
while (b = 10) { }
```

[https://en.wikipedia.org/wiki/Side_effect_\(computer_science\)](https://en.wikipedia.org/wiki/Side_effect_(computer_science))

Pure Languages

Haskell is a **pure** language

programs are made of **functions**


that can't change any global state or variables,

they can only do some computations and return them results.

every variable's value does not change in time

However, some problems are inherently stateful

in that they rely on some state that changes over time.

`st1 = 10` 

`s` \rightarrow `(x,s)`

`st1` `(v,10)`

a bit tedious to model

Haskell has the **state monad** features

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

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:

$s \rightarrow (a, s)$

s is the type of the **state** and **a** the **result** of the **stateful computation**.

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

Assignment as a stateful computation

Assignment in other languages

x = 5

could be thought of as a **stateful computation**.

in an imperative language :

will assign the value **5** to the variable **x**

will have the value **5** as an *expression*

in a functional language

as a **function** that takes a **state** and returns a **result** and a **new state**

an input **state** :

all the variables that have been assigned previously

a **result** : **5**

a **new state** :

all the previous variable mappings plus the newly assigned variable.

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

A value with a context

The stateful computation:

- a **function** that takes a **state** and returns a **result** and a **new state**
- can be considered as a **value** with a **context**

the actual **value** is

the **result**

the **context** is

that we have to provide some **initial state** to get the result and
that apart from getting the result we also get a **new state**.

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

Side Effects in Haskell

We give it the **World**

we got back the **World**

from getting **x** out of its monad,

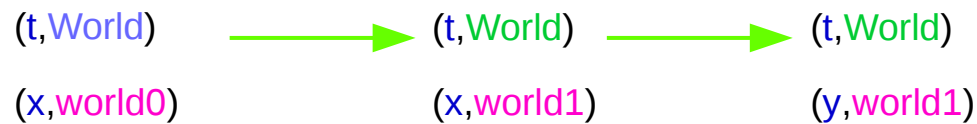
and the thing it gives back to us is

the **y** with

a final version of the **World**

.

the implementation of bind



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

Side Effects in Haskell

Which World was given initially?
Which World was updated?

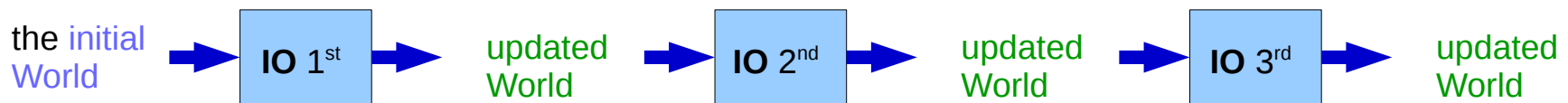
In **GHC**, a **main** must be defined somewhere with type **IO ()**

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>

Side Effects in Haskell

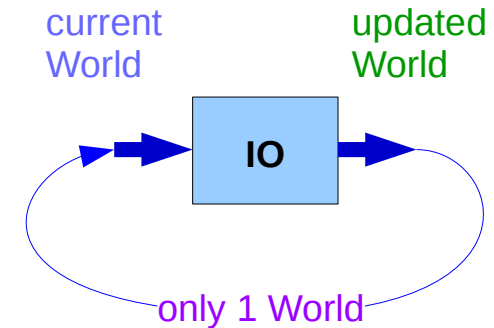
when using **GHCI**,
everything is wrapped in **an implicit IO**,
since the results get printed out to the screen.

Every time a new command is given to GHCI,
GHCI passes **the current World**,
GHCI gets the *result* of the command back,
GHCI request to display the *result*
(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),

and then saves **the new World** to give to the next command.

the implementation of bind



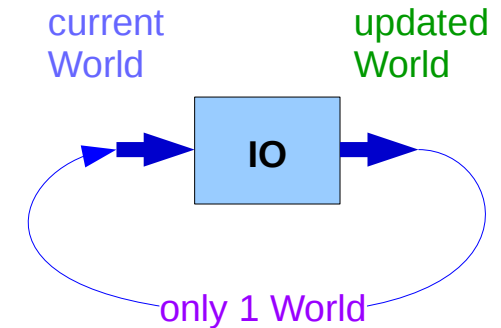
<https://www.cs.hmc.edu/~adavidso/monads.pdf>

Side Effects in Haskell

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 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>

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://stackoverflow.com/questions/2488646/why-are-side-effects-modeled-as-monads-in-haskell>

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

Monadic Operation

Monadic operations tend to have types which look like

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

where the **return type** is a type application:

the function tells you

which **effects** are possible

and the argument tells you (val-out-type)

what sort of **value** is produced **by the operation**

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

```
putStr :: String -> IO ()
```

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

Monadic Operation – put, putStr

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

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

one value input type **s**

the **effect-monad State s**

the **value output type ()**

the operation is used *only for its effect*;

the value delivered is uninteresting

```
putStr :: String -> IO ()
```

delivers a string to stdout but does not return anything exciting.

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

Side Effects of IO Monad

Generally, a monad cannot perform **side effects** in Haskell.
there is one exception: **IO monad**

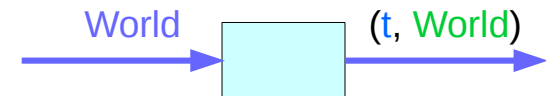
Suppose there is a **type** called **World**,
which contains all the state of the external universe

A way of thinking what **IO monad** does

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

```
putStr :: String -> IO ()
```

```
World -> (t, World)
```



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

Type Synonym **IO t**

IO t is a parameterized function

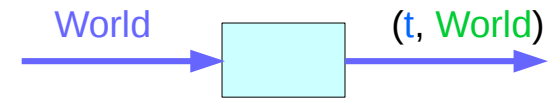
input : a **World**

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

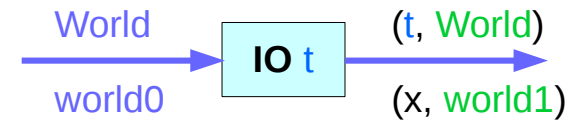
```
type IO t = World -> (t, World)
```

type synonym

World -> (t, World)



IO t



(x, world1) :: IO t World

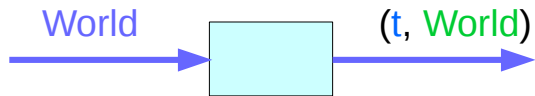
<https://www.cs.hmc.edu/~adavidso/monads.pdf>

Side Effects of IO Monad

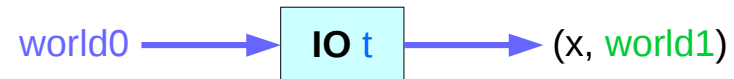
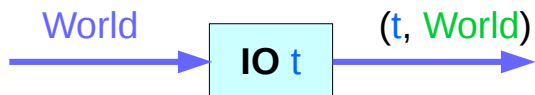
```
type IO t = World -> (t, World)
```

type synonym

```
World -> (t, World)
```



```
IO t
```



```
(x, world1) :: IO t World
```

```
x :: t
```

```
world1 :: World
```

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

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

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

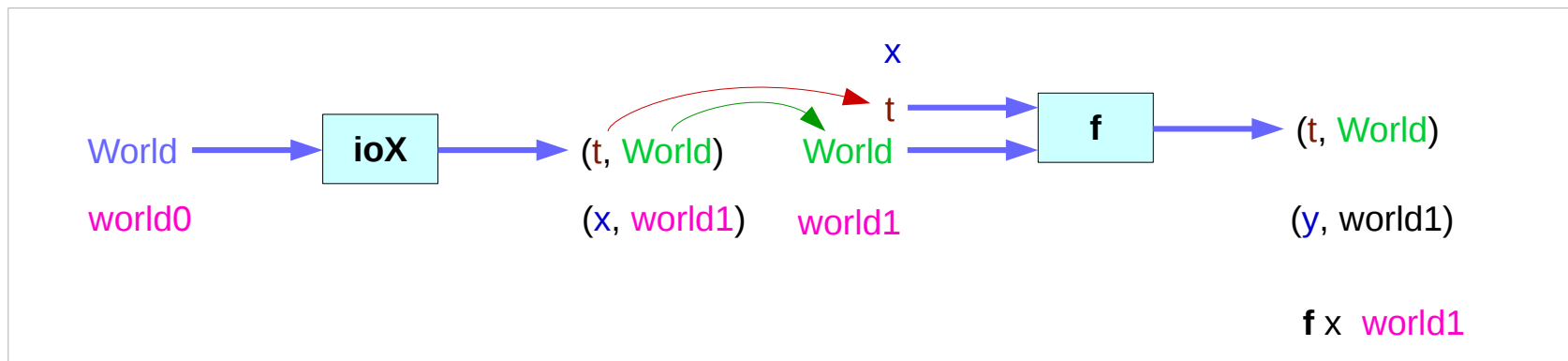
IO Monad Implementation

```
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)
```

$(x, s) \longrightarrow (x, s') \longrightarrow (y, s')$

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



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

Monad IO and Monad ST

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)
```

instance Monad ST where

```
-- return :: a -> ST a
```

```
return x = \s -> (x,s)
```

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

```
st >>= f = \s -> let (x,s') = st s
                  in f x s'
```

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

type synonym

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

State Transformers ST

instance **Monad ST** where

```
-- return :: a -> ST a
```

```
return x = \s -> (x,s)
```

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

```
st >>= f = \s -> let (x,s') = st s in f x s'
```

>>= provides a means of sequencing **state transformers**:

st >>= f applies the **state transformer st** to an initial state **s**,

then applies the function **f** to the resulting value **x**

to give a second **state transformer (f x)**,

which is then applied to the modified state **s'** to give the final result:

```
st >>= f = \s -> f x s'
```

```
where (x,s') = st s
```

```
st >>= f = \s -> (y,s')
```

```
where (x,s') = st s
```

```
(y,s') = f x s'
```

```
(x,s') = st s
```

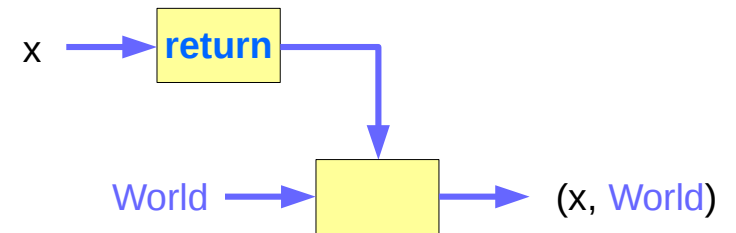
```
f x s'
```

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

Monad IO - return

The **return** function takes x
and gives back a function
that takes a **World**
and returns x along with the new, **updated World (=World)**
formed by not modifying the **World** it was given

return x world = (x, world)



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

Monad IO - >>=

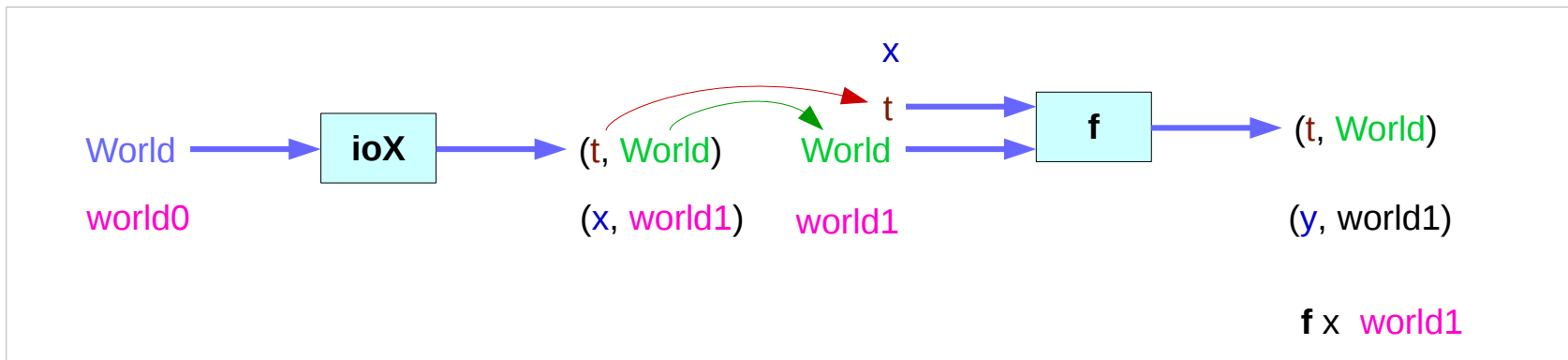
the expression $(\text{ioX} \gg= \text{f})$ has

type $\text{World} \rightarrow (\text{t}, \text{World})$

a function **ioX** that takes **world0** of the type **World**,
which is used to extract **x** from its **IO** monad.

x gets passed to **f**, resulting in another **IO** monad,
which again is a function that takes **world1** of the type **World**
and returns a **y** and a new, updated **World**.

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>