Monad Background (3A)

Young Won Lim
11/8/17

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

This document was produced by using OpenOffice.

Young Won Lim
11/8/17

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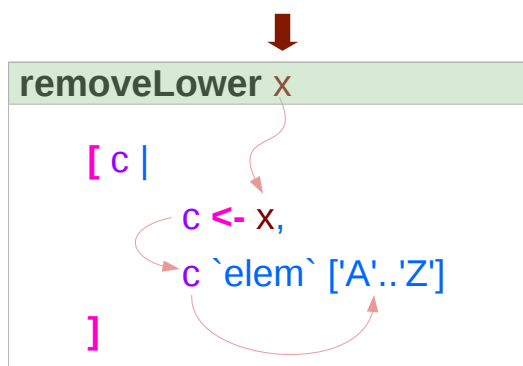
Haskell in 5 steps
https://wiki.haskell.org/Haskell_in_5_steps

A List Comprehension Function

let **removeLower** x = [c | c **<-** x, c `elem` ['A'..'Z']]

a **list comprehension**

removeLower x

[c |

    c **<-** x,

    c `elem` ['A'..'Z']

]

"Hello"

[c: 'H'

  c: 'e'

  c: 'l'

  c: 'l'

  c: 'o']

"H"

do { x1 **<-** action1

   ; x2 **<-** action2

   ; mk_action3 x1 x2 **}**

x1 : Return value of action1

x2: Return value of action2

https://stackoverflow.com/questions/35198897/does-mean-assigning-a-variable-in-haskell

Pattern and Predicate

let removeLower x = [c | c <- x, c `elem` ['A'..'Z']]

a list comprehension

[c | c <- x, c `elem` ['A'..'Z']]

    c <- x is a **generator**

        (x : argument of the function **removeLower**)

    c is a **pattern**

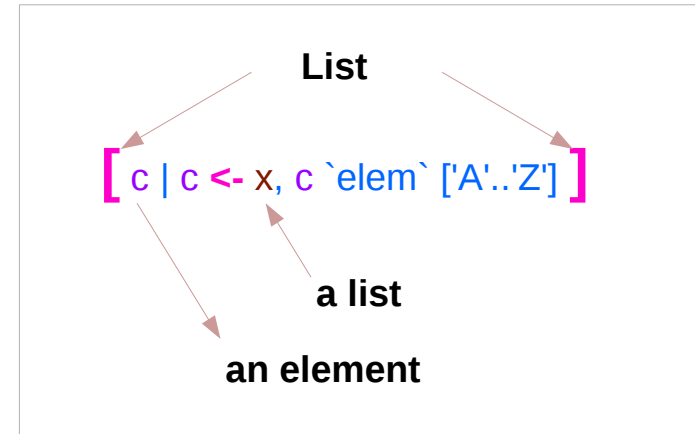   matching from the **elements** of the **list x**

        successive binding of c to the **elements** of the **list x**

    c `elem` ['A'..'Z']

        is a **predicate** which is applied to each successive binding of c

        Only c which passes this predicate will appear in the output list

List

[c | c <- x, c `elem` ['A'..'Z']]

a list

an element

Assignment in Haskell

Assignment in Haskell : <u>declaration</u> with <u>initialization</u>:

- no uninitialized variables,
- must declare with <u>an initial value</u>
- <u>no</u> <u>mutation</u>
- a variable keeps its initial value throughout its scope.

https://stackoverflow.com/questions/35198897/does-mean-assigning-a-variable-in-haskell

Young Won Lim
11/8/17

Generator

[c| c **<-** x, c `elem` ['A'..'Z']]

filter (`elem` ['A' .. 'Z']) x

[c | c **<-** x **]**

c: an element
x: a list

do c **<-** x
   return c

x **>>=** (\c -> **return** c)

c: an element   or   c: a list
x: an element   x: a list

x **>>= return**

action1 **>>= (**\ x1 ->
  action2 **>>= (**\ x2 ->
    mk_action3 x1 x2 **))**

https://stackoverflow.com/questions/35198897/does-mean-assigning-a-variable-in-haskell

Anonymous Functions

(\x -> x + 1) 4
5 :: Integer

(\x y -> x + y) 3 5
8 :: Integer

inc1 = \x -> x + 1

---

incListA lst = **map inc2** lst
   where **inc2** x = x + 1

---

incListB lst = **map** (\x -> x + 1) lst

---

incListC = **map** (+1)

---

https://wiki.haskell.org/Anonymous_function

Then Operator (**>>**) and **do** Statements

a <u>chain</u> of actions

to *<u>sequence</u>* input / output operations

the (**>>**) (**then**) operator works almost identically in **do** notation
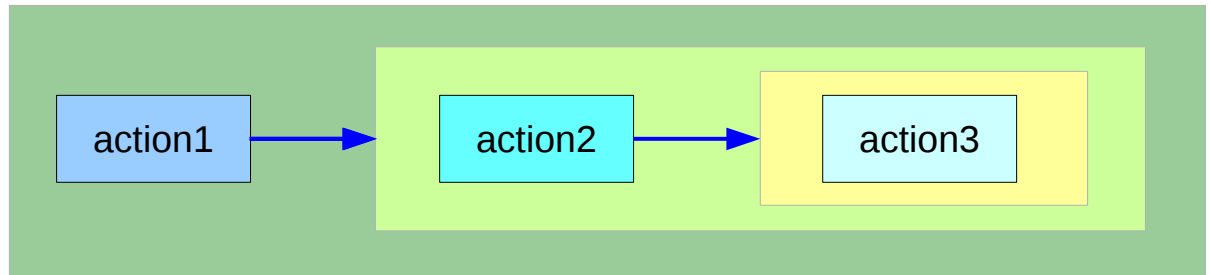
```
putStr "Hello"  >>
putStr " "         >>
putStr "world!" >>
putStr "\n"
```

```
do { putStr "Hello"
   ; putStr " "
   ; putStr "world!"
   ; putStr "\n" }
```

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

Chaining in **do** and **>>** notations

```
do { action1
   ; action2
   ; action3 }
```



```
do { action1
   ; do { action2
        ; action3 } }
```

➡

```
action1 >>
do { action2
   ; action3 }
```

can **chain** any actions
all of which are in **the same monad**

```
do { action1
   ; do { action2
        ; do { action3 } } }
```

➡

```
action1 >>
  action2  >>
    action3
```

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

Bind Operator (**>==**) and **do** statements

The bind operator (**>>=**)

    passes a value **->**

    (the result of an action or function),

    downstream in the binding sequence.

do notation assigns a variable name

to the passed value using the **<-**

```
action1 >>= (\ x1 ->
  action2 >>= (\ x2 ->
   mk_action3 x1 x2 ))
```

```
do { x1 <- action1
   ; x2 <- action2
   ; mk_action3 x1 x2 }
```

anonymous function
(lambda expression)
is used

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

Chaining **>>=** and **do** notations

->

action1 **>>=** (\ x1 **->** action2 **>>=** (\ x2 **->** mk_action3 x1 x2 **))**

action1
  >>=
    (\ x1 **->** action2
      >>=
        (\ x2 **->** mk_action3 x1 x2 **))**

action1 **>>=** (\ x1 ->
  action2 **>>=** (\ x2 ->
    mk_action3 x1 x2 **))**

<-

do { x1 **<-** action1
    ; x2 **<-** action2
    ; mk_action3 x1 x2 **}**
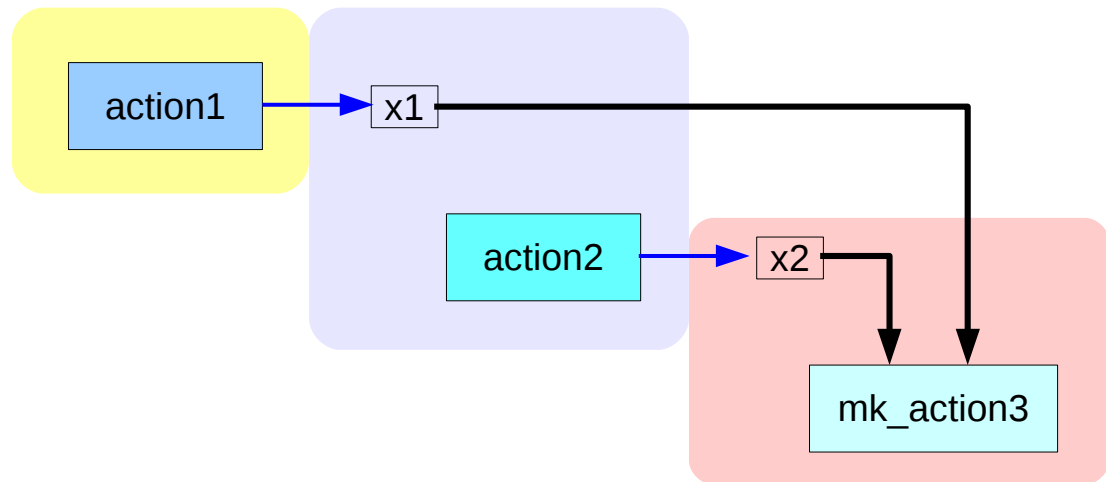
https://en.wikibooks.org/wiki/Haskell/do_notation

fail method

```
do { Just x1 <- action1
   ;        x2 <- action2
   ; mk_action3 x1 x2   }
```

```
do { x1 <- action1
   ; x2 <- action2
   ; mk_action3 x1 x2 }
```

O.K. when action1 returns Just x1

when action1 returns Nothing
crash with an non-exhaustive patterns error

Handling failure with **fail** method

```
action1 >>= f where
   f (Just x1) = do { x2 <- action2
                    ; mk_action3 x1 x2 }
   f _         = fail "..."
```

-- A compiler-generated message.

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

Example

```
nameDo :: IO ()
nameDo = do { putStr "What is your first name? "
            ; first <- getLine
            ; putStr "And your last name? "
            ; last <- getLine
            ; let full = first ++ " " ++ last
            ; putStrLn ("Pleased to meet you, " ++ full ++ "!") }
```

```
do { x1 <- action1
   ; x2 <- action2
   ; mk_action3 x1 x2 }
```

using the **do** statement

A possible translation into vanilla monadic code:

```
nameLambda :: IO ()
nameLambda = putStr "What is your first name? " >>
             getLine >>= \ first ->
             putStr "And your last name? " >>
             getLine >>= \ last ->
             let full = first ++ " " ++ last
             in putStrLn ("Pleased to meet you, " ++ full ++ "!")
```

using **then (>>)** and **Bind (>>=)** operators

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

return method

```
nameReturn :: IO String

nameReturn = do putStr "What is your first name? "
                first <- getLine
                putStr "And your last name? "
                last <- getLine
                let full = first ++ " " ++ last
                putStrLn ("Pleased to meet you, " ++ full ++ "!")
                return full
```

```
greetAndSeeYou :: IO ()

greetAndSeeYou = do name <- nameReturn
                    putStrLn ("See you, " ++ name ++ "!")
```

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

Young Won Lim
11/8/17

Without a **return** method

```
nameReturn :: IO String
nameReturn = do putStr "What is your first name? "
                first <- getLine
                putStr "And your last name? "
                last <- getLine
                let full = first ++ " " ++ last
                putStrLn ("Pleased to meet you, " ++ full ++ "!")
                return full
```

explicit return statement

returns **IO String** monad

```
nameDo :: IO ()
nameDo = do { putStr "What is your first name? "
            ; first <- getLine
            ; putStr "And your last name? "
            ; last <- getLine
            ; let full = first ++ " " ++ last
            ; putStrLn ("Pleased to meet you, " ++ full ++ "!") }
```

no return statement

returns **empty IO** monad

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

return method – not a final statement

```
nameReturnAndCarryOn :: IO ()
nameReturnAndCarryOn = do putStr "What is your first name? "
                          first <- getLine
                          putStr "And your last name? "
                          last <- getLine
                          let full = first++" "++last
                          putStrLn ("Pleased to meet you, "++full++"!")
                          return full
                          putStrLn "I am not finished yet!"
```

the return statement does <u>not</u> interrupt the flow

the last statements of the sequence returns a value

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

Data Constructor

data **Color** = **Red** | **Green** | **Blue**

      Color   is a type

      Red   is a _constructor_ that contains a _value_ of type **Color**.
      Green   is a _constructor_ that contains a _value_ of type **Color**.
      Blue   is a _constructor_ that contains a _value_ of type **Color**.

https://stackoverflow.com/questions/18204308/haskell-type-vs-data-constructor

Data Constructor with Parameters

data **Color** = **RGB** Int Int Int

      Color   is a type

      RGB   is not a value but a *function* taking three Int's and *returning* *a* *value*

      RGB :: Int -> Int -> Int -> Color

            RGB is a **data constructor** that is a *function*

                  taking three Int values as its arguments,

                  and then uses them to construct a new value.

Type Constructor

Consider a binary tree to store Strings

data **SBTree** = **Leaf** String | **Branch** String **SBTree SBTree**

a type

SBTree	is a **type**
Leaf	is a **data constructor** (a function)
Branch	is a **data constructor** (a function)

Leaf :: String -> SBTree

Branch :: String -> SBTree -> SBTree -> SBTree

Young Won Lim
11/8/17

Similar Type Constructors

Consider a binary tree to store Strings

data **SBTree** = **Leaf** String | **Branch** String **SBTree SBTree**

Consider a binary tree to store Bool

data **BBTree** = **Leaf** Bool | **Branch** Bool **BBTree BBTree**

Consider a binary tree to store a parameter type

data **BTree** a = **Leaf** a | **Branch** a (**BTree** a) (**BTree** a)

Type Constructor with a Parameter

Type constructors

Both **SBTree** and **BBTree** are type constructors

data **SBTree** = **Leaf** String | **Branch** String **SBTree SBTree**
data **BBTree** = **Leaf** Bool | **Branch** Bool **BBTree BBTree**

data **BTree** a = **Leaf** a | **Branch** a (**BTree** a) (**BTree** a)

Now we introduce a type variable a as a parameter to the type constructor.

BTree has become a function.
It takes a type as its argument and it returns a new type.

https://stackoverflow.com/questions/18204308/haskell-type-vs-data-constructor

Type Constructors and Data Constructors

A **type constructor**

- a "function" that takes 0 or more types
- gives you back <mark>a new **type**</mark>.

Type constructors with parameters

   allows slight variations in types

```
type SBTree = BTree String
type BBTree = BTree Bool
```

A **data constructor**

- a "function" that takes 0 or more values
- gives you back <mark>a new **value**</mark>.

Data constructors with parameters

   allows slight variations in values

```
RGB 12 92 27
#0c5c1b
RGB 255 0 0
RGB 0 255 0
RGB 0 0 255
```

https://stackoverflow.com/questions/18204308/haskell-type-vs-data-constructor

( )

( ) is both a **type** and a **value**.

( ) is a special **type**, pronounced "unit",
has one **value** ( ), sometimes pronounced "void"

the **unit type** has only one **value** which is called **unit**.

( ) :: ( )   **Type** :: **Expression**

It is the same as the void type **void** in Java or C/C++.

Unit Type

a **unit type** is a type that allows *only one value* (and thus can hold *no information*).

It is the same as the void type **void** in Java or C/C++.

```
:t
Expression :: Type
```

```
data Unit = Unit

Prelude> :t Unit
Unit :: Unit
```

```
Prelude> :t ()
() :: ()
```

Type Language and Expression Language

data **T**const **T**var … **T**var = **V**const type … type | …
                                     Vconst type … type

A new datatype declaration

Tconst (Type Constructor)   is added to *the type language*

Vconst (Value Constructor)   is added to *the expression language* and *its pattern sublanguage*
                                     must not appear in types

Argument types in **V**const type … type
    are the types given to the arguments (**T**const **T**var … **T**var)
    are used in expressions

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

Datatype Declaration Examples

data **Tree** a =   **Leaf** | **Node** (Tree a) (Tree a)

Tree   (Type Constructor)
Leaf or **Node**   (Value Constructor)

data **Type** = **Value**

data **( )** = **( )**

( )   (Type Constructor)
( )   (Value Constructor)

the type (), often pronounced "Unit"
the value (), sometimes pronounced "void"

the type () containing only one value ()

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

IO ( )

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 what sort of value

is produced by the operation

Young Won Lim
11/8/17

IO ( ) - Effect Monad

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.

Functional & 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

r = 5

r = 2

---

imperative programming:

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

Hakell 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.

Immutable: They vary 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

r = 5

r = 2

No mutation

In Haskell

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   Var2.hs

r = 5   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

r = r + 1

imperative programming:

incrementing the variable r

(updating the value in memory)

Hakell 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

x = 3   y = x * 3

Hakell 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 area r = ... by the RHS ... = pi * r^2 }

pi * 5 ^ 2

=> { replace pi by its numerical value }

3.141592653589793 * 5 ^ 2

=> { apply exponentiation (^) }

3.141592653589793 * 25

=> { apply multiplication (*) }

78.53981633974483

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 of IO Monad

Generally, a monad <u>cannot</u> perform side effects in Haskell.

there is one exception: **IO monad**

Suppose there is a type called World,

which contains all the <u>state</u> of the external universe
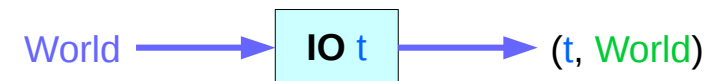
A way of thinking what IO monad does

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

type synonym

World -> (t, World)

World → [   ] → (t, World)

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

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.

World → **IO** t → (t, World)

IO x world0   (x, world1)

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

Type Synonym **IO t**

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

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.

It is <u>impossible</u> to store the extra copies of the contents of your hard drive
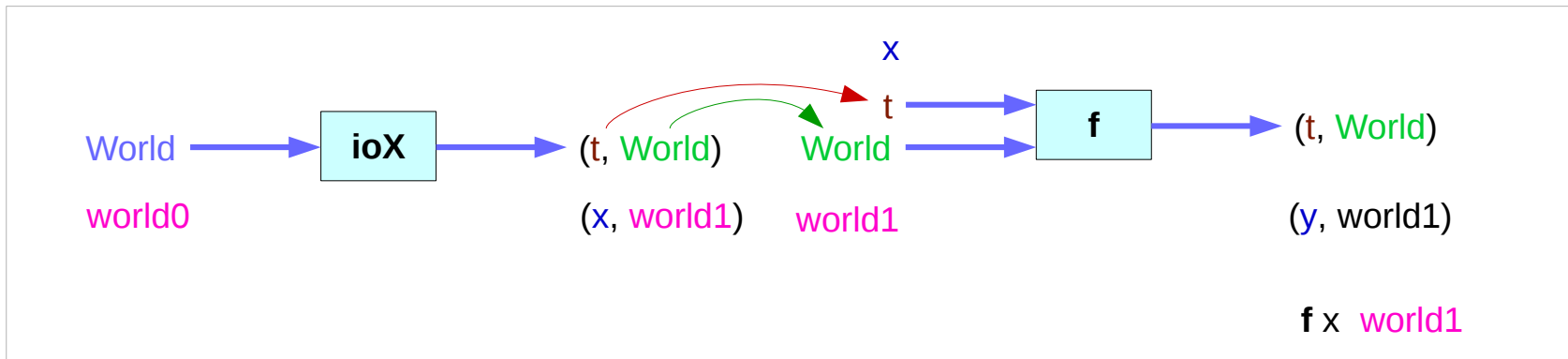that each of the Worlds contains

given World → updated World

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

x

World   →   **ioX**   →   (t, World)   t   →   **f**   →   (t, World)

                            World

world0   (x, world1)   world1   (y, world1)

                                                 f x world1

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
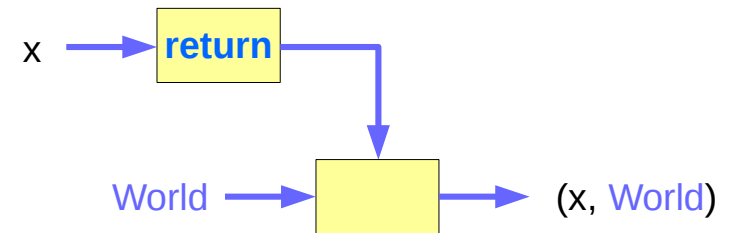
f x s'

Monad IO - return

The **return** function takes x

and gives back a <u>function</u>

  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)

x → **return** →

World → □ → (x, World)

Monad IO - >>=

the expression (**ioX >>= f**) has

type World -> (t, World)

     a function **ioX** that takes world0 of the type World,

     which is used to <u>extract</u> 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 <u>new</u>, <u>updated</u> World.

the implementation of bind

https://www.cs.hmc.edu/~adavidso/monads.pdf
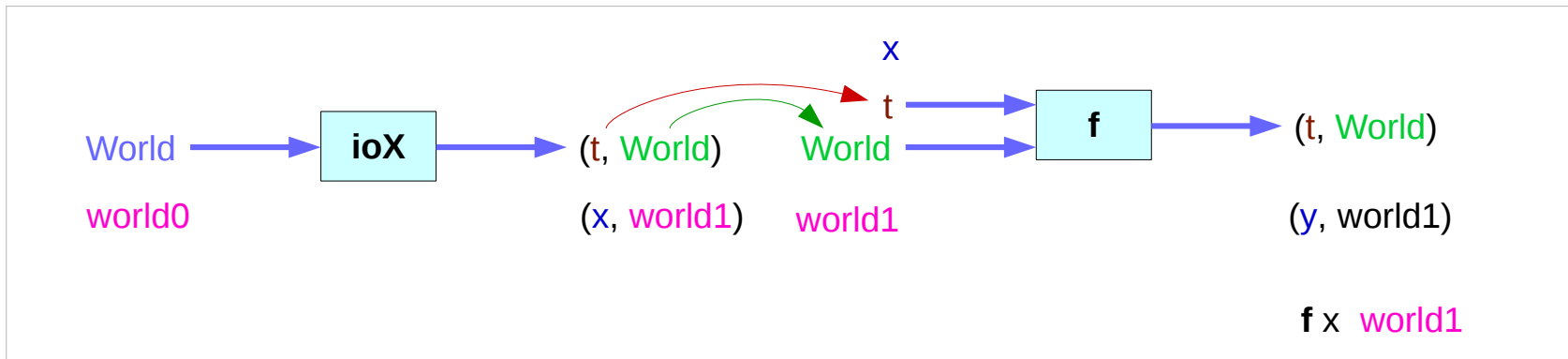
Side Effects in Haskell

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

(t,World) ⟶ (t,World) ⟶ (t,World)

(x,world0) 　　　(x,world1) 　　　(y,world1)

https://www.cs.hmc.edu/~adavidso/monads.pdf
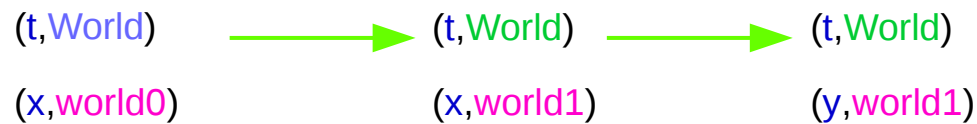
Side Effects in Haskell

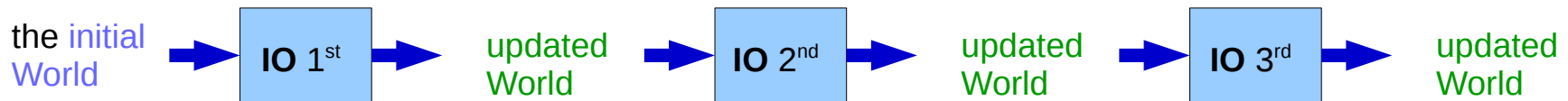Which World was given initially?

Which World was updated?

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

a program execution <u>starts</u> 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 <u>not</u> <u>reachable</u> from **main** will <u>never</u> <u>be</u> <u>executed</u>

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

The modification of the World

the initial World → **IO 1st** → updated World → **IO 2nd** → updated World → **IO 3rd** → updated World

https://www.cs.hmc.edu/~adavidso/monads.pdf
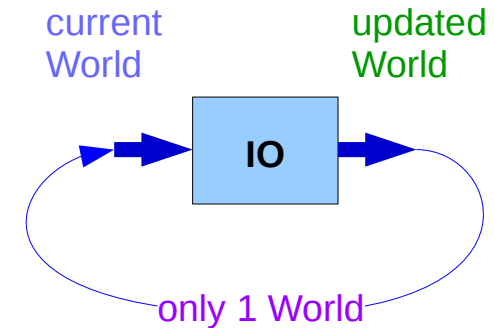
Side Effects in Haskell

when using **GHCI**,

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

since the results get printed out to the screen.

Every time a <u>new</u> <u>command</u> 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 <u>next</u> <u>command</u>.

the implementation of bind

current World   updated World

IO

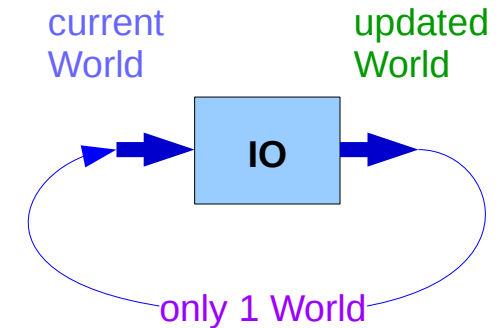only 1 World

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.

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

current World   updated World

IO

only 1 World

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