Monad P2 : State Transformer Basics (1A)

Young Won Lim
10/2/19

Copyright (c) 2016 - 2019 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 LibreOffice.

Based on

Haskell in 5 steps

https://wiki.haskell.org/Haskell_in_5_steps

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

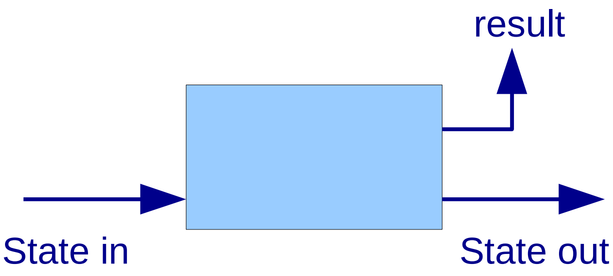
https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/state-lasc.pdf

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

A state transformer – a pure function

A **state transformer** of type (**ST s a**) is a **computation**
which transforms a **state** indexed by type **s** ,
and delivers a **value** of type a .

You can think of it as a **pure function**,
taking a **state** as its argument,
and delivering a **state** and a **value** as its result.

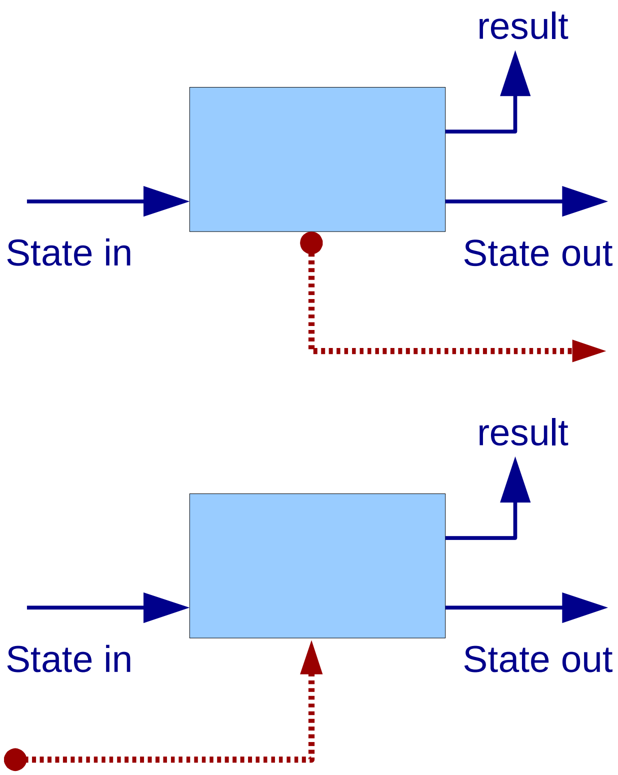


https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/state-lasc.pdf

A state transformer : a first-class value

From a semantic point of view,
this is a purely-functional account of state.
being a **pure function**,
a **state transformer** is a **first-class value**:

it can be passed to a function,
returned as a result,
stored in a data structure,
duplicated freely, and so on.

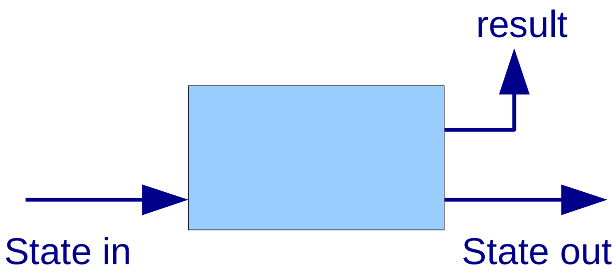


https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/state-lasc.pdf

A state transformer – a stateful computation

we take the term **state transformer** to be synonymous with **stateful computation**:

the **computation** is seen as **transforming** one **state** into **another**.



state transformer

stateful computation

https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/state-lasc.pdf

A State Transformer – a functional type and a tuple

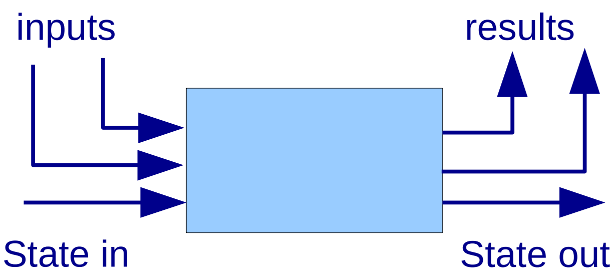
A state transformer can have
other inputs besides the state;
if so, it will have a **functional type**.

It can also have **many results**,
by returning them in a **tuple**.

a **state transformer** with two **inputs** of type **Int** ,
and two **results** of type **Int** and **Bool**

$\underbrace{\textbf{Int -> Int ->}}$ **ST s (Int,Bool)**

functional type



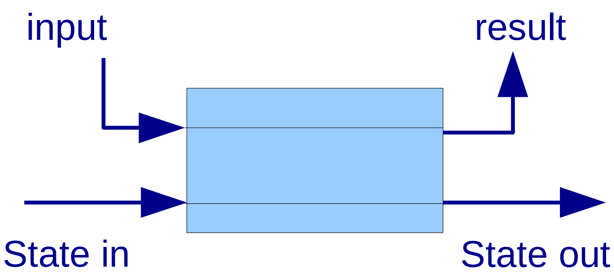
https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/state-lasc.pdf

A state transformer – **returnST**

The simplest **state transformer**, **returnST** ,

simply delivers a **value** without a affecting the **state** at all:

returnST :: a -> ST s a



https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/state-lasc.pdf

A Monad Transformer

Monad Transformers:

special types that allow us to roll two monads into
a single one that shares the behavior of both.

MaybeT define a monad transformer that gives the **IO** monad
some characteristics of the **Maybe** monad

Precursor monad refers to
the non-transformer monad (e.g. **Maybe** in **MaybeT**)
on which a transformer is based

Base monad refers to the other monad
(e.g. **IO** in MaybeT IO) on which the transformer is applied.

IO (Maybe String)

MaybeT IO String

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

Some Monad Transformer Examples

Precursor	**Transformer**	**Original Type by precursor**	**Combined Type by transformer**
Writer	**WriterT**	**(a, w)**	**m (a, w)**
Reader	**ReaderT**	**r -> a**	**r -> m a**
State	**StateT**	**s -> (a, s)**	**s -> m (a, s)**
Cont	**ContT**	**(a -> r) -> r**	**(a -> m r) -> m r**

IO (Maybe String)

MaybeT IO String

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

A State Transformer (ST)

```
type State = ...

type ST = State -> State                    -- a function type
```

about functions that manipulate some kind of state

this state can be represented by a type (State)

a **state transformer** **ST**   a state manipulating function

takes the current state as its **argument**

produces a modified state as its **result**

which reflects any side effects performed by the function:

a **state transformer** **ST**
not **Monad Transformer**

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

A Generalized State Transformer

```
type State = ...

type ST = State -> State

type ST a = State -> (a, State)
```

generalized state transformers

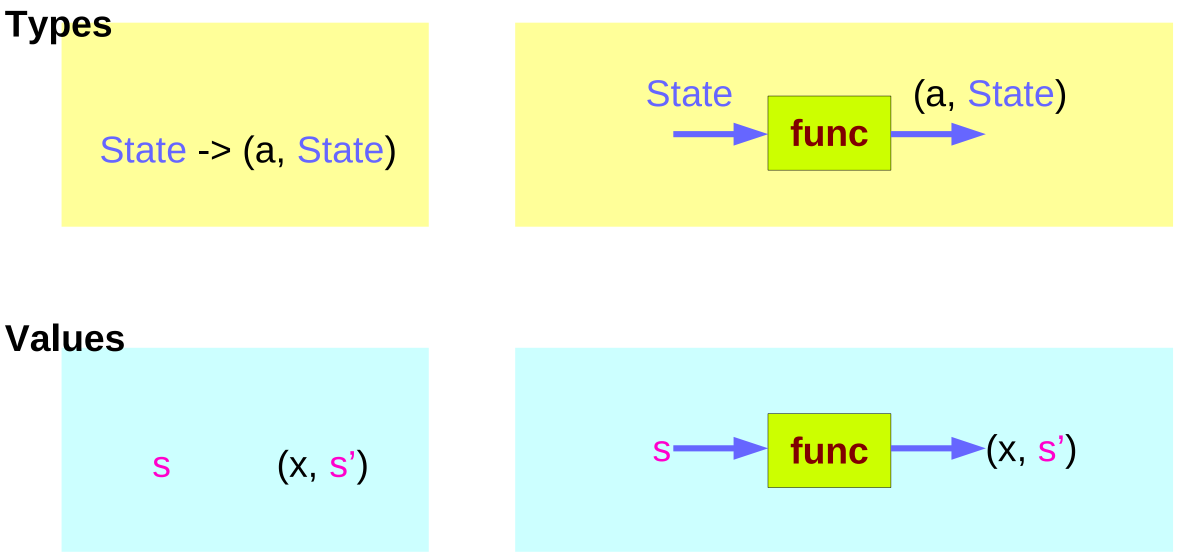
return a result value in addition to the modified state

specify the result type as a parameter of the **ST** type

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

Types and Values

```
type ST a = State -> (a, State)
```



Types

State -> (a, State)

State → **func** → (a, State)

Values

s (x, s')

s → **func** → (x, s')

A function is also a value

```
func :: ST a
func :: State -> (a, State)
```

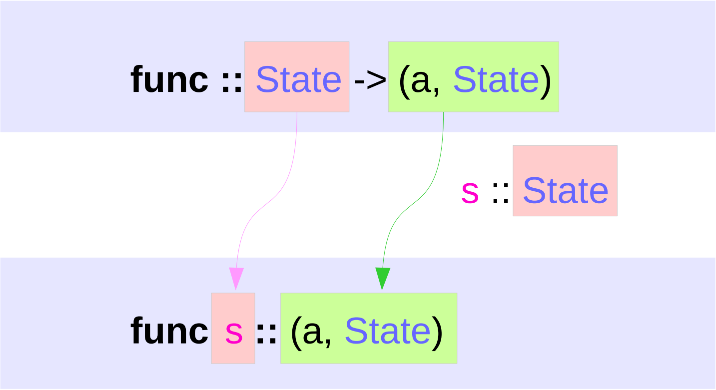
x :: a	the result value
s :: State	input state value
s' :: State	output state value

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

func and func s type signatures

type **ST** a = State -> (a, State)

func :: **ST** a



func :: State -> (a, State)

s :: State

func s **::** (a, State)

func s ➡ (x, s')

application of
input s gives
output (x, s')

func :: **ST** a
x :: a
s :: State
s' :: State

func :: **ST** a
func :: State -> (a, State)
func s ➡ (x, s')
func s :: (a, State)

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

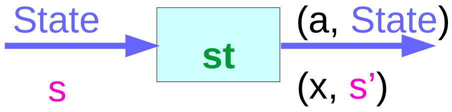
Function input and output types

type **ST** a = State -> (a, State)

st s ⇨ (x, s')

generalized state transformer

(a_result, updated_state) :: (a, State)



st :: **ST** a
s :: State
(x, s') :: (a, State)

st :: **ST** a
s :: State
st s :: (a, State)

st s ⇨ (x, s')

type **ST** a = State -> (a, State)

st s ⇨ (x, s')

application of
input s gives
output (x, s')

st s ⇨ (x, s')

st s :: ~~**ST** a State~~ **st** s :: (a, State)

(x, s') :: ~~**ST** a State~~ (x, s') :: (a, State)

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

Taking an additional argument

type ST Int = State -> (Int, State)

How to convert **ST Int** into a state transformer

that takes a character and returns an integer ?

further generalization of the state transformer **ST**

which takes an argument of type b

- no need to use more generalized ST type
- instead, use currying.

type ST2 a b

type ST3 b a

type ST2 a b = b -> State -> (a, State)

type ST3 b a = b -> State -> (a, State)

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

A Curried Generalized State Transformer

type ST a = State **-> (a,** State**)** generalized ST

type ST3 b a = b -> State **-> (a,** State**)** further generalized ST

b -> ST a = b -> State **-> (a,** State**)** think currying

a state transformer

that takes a character

and returns an integer

would have type **Char -> ST Int**

Char -> State -> (**Int**, State) **curried form**

* Curried Function

f x y	**f :: a -> b -> c**
(f x) y	**f :: a -> (b -> c)**

f x returns a function of type **b -> c**

g y	**g :: b -> c**

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

ST Monad Instance – return method

```
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'
```

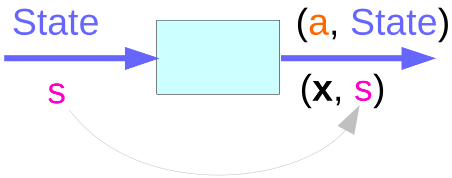
type ST …. ~~instances~~ (X)
data ST … instances (O)

ST : an instance of a monadic type

return converts a value (**x**)
into a state transformer (s ->(**x**,s))
that simply *returns that value* (**x**)
without modifying the state (s → s)

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

a function is a value



return x returns
a value of **ST a** type

to execute this function
an argument to s is necessary

ST Monad Instance – >>= method

```
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'
```

```
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'
```

sequencing state transformers: **st >>= f**

- the 1st state transformer **st** — **st** s ➡ (x,s') — (1) **input monad** (update + compute)
- the 2nd state transformer (**f** x) — **f** x s' ➡ (y,s') — (2) **return monad** (result argument)

1) apply **st** to an initial state s, to get (x,s')

2) apply the function **f** to the x, the value of result

3) apply (**f** x) to the updated state s'

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

The type signatures of the sequencer >>=

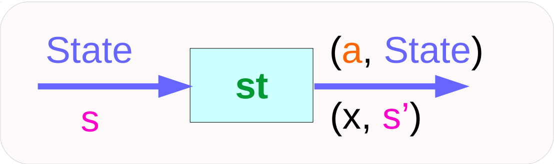
```
instance Monad ST where
  -- return :: a -> ST a
  return x  =  \s -> (x,s)

  -- (>>=)  :: ST a -> (a -> ST b) -> ST b
 st >>= f  =  \s -> let (x,s') = st s in f x s'
```

st :: ST a
f :: a -> ST b
(>>=) :: ST a -> (a -> ST b) -> ST b

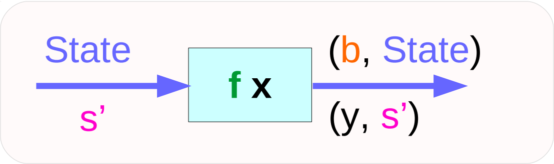
st :: State -> (a, State)
f :: a -> State -> (b, State)
(>>=) :: State -> (a, State) -> (a -> State -> (b, State)) -> State -> (b, State)

type ST a = State -> (a, State)

st :: ST a



f x :: ST a



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

The type of st s and f x s'

st :: State **->** (a, State)
f :: a -> State **->** (b, State)
(>>=) :: State **->** (a, State) **->** (a **->** State **->** (b, State)) **->** State **->** (b, State)

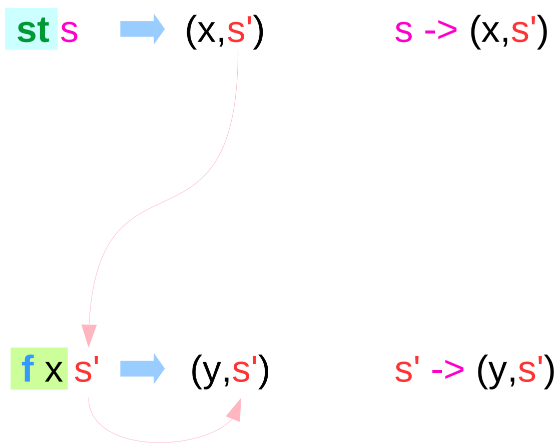
st :: State **->** (a, State)

st s :: (a, State)

f :: **a ->** State **->** (b, State)

f x :: State **->** (b, State)

f x s' :: (b, State)



st s ➡ (x,s')   s -> (x,s')

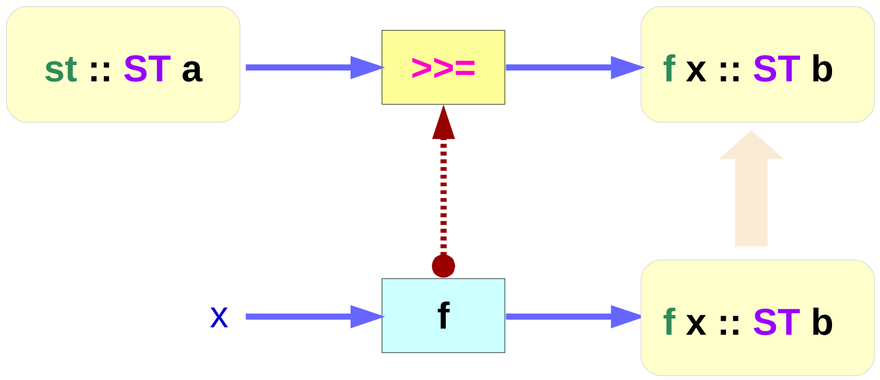
f x s' ➡ (y,s')   s' -> (y,s')

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

ST Monad - (>>=) operator type diagram

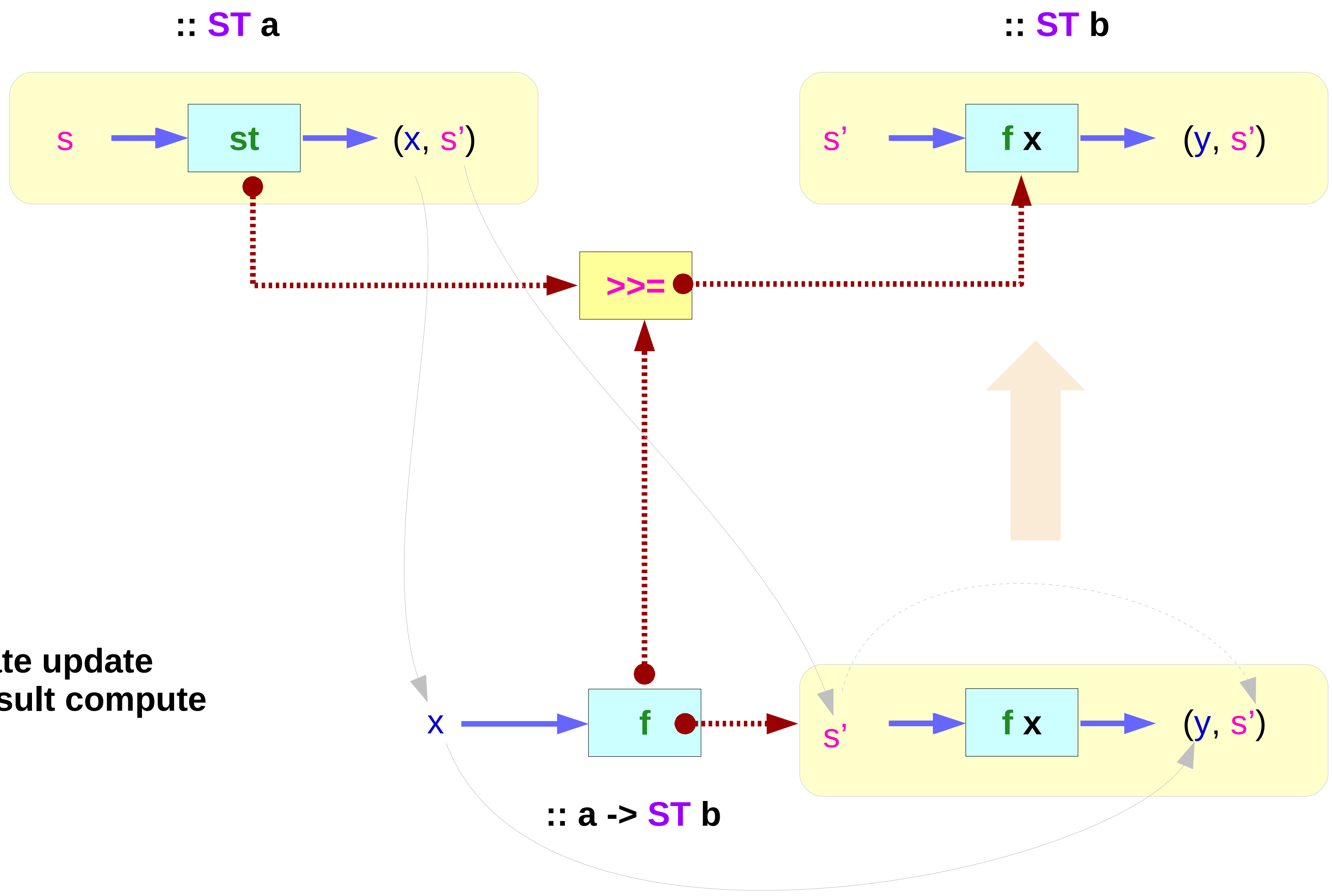
monad	monad returning function	monad
1st arg	2nd arg	return
ST a	(a -> ST b)	ST b

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



st – state update
f x – result compute

ST Monad - (**>>=**) execution of **st** & **f x**

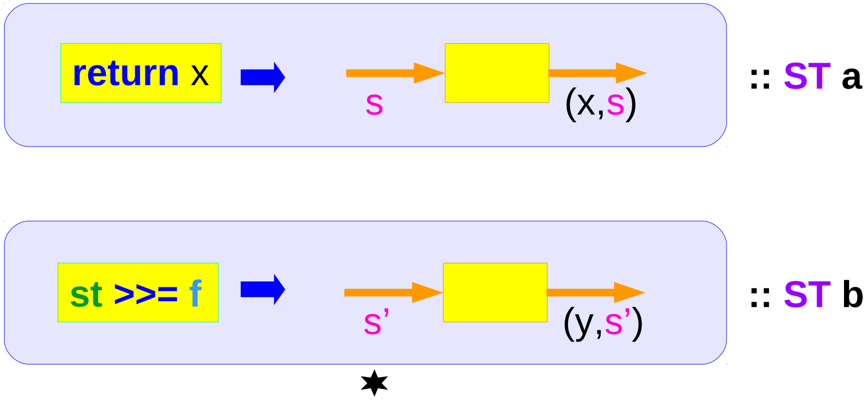


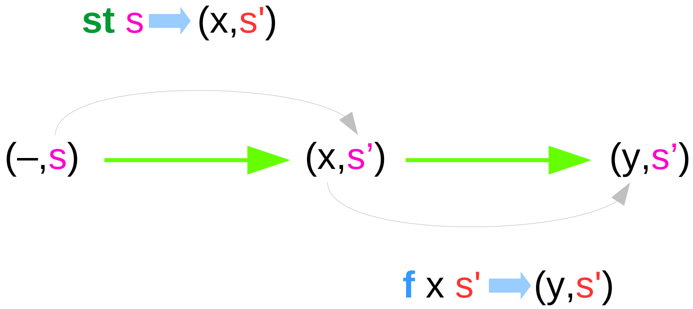
st – **state update**
f x – **result compute**

ST Monad – return and >>=

```
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'
```

a function is a value





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

List, Maybe, and ST Monads

```
instance Monad [] where
   -- return :: a -> [a]
   return x  =  [x]

   -- (>>=)  :: [a] -> (a -> [b]) -> [b]
   xs >>= f  =  concat (map f xs)
```

```
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'
```

```
instance Monad Maybe where
   -- return:: a->Maybe a
   return x       =  Just x
   -- (>>=) ::
   Maybe a -> (a -> Maybe b) -> M aybe b
   Nothing  >>= _ =  Nothing
   (Just x)  >>= f =  f x
```

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

Dummy Constructor **DC**

type **ST** a = State -> (a, State) ~~instances~~, ~~constructor~~ (X)

data **ST0** a = **DC** (State -> (a, State)) instances, constructor (O)

to make instances

use the **data** mechanism instead of **type**

with a **dummy constructor** (DC)

pattern matching purpose – any name is ok

data **ST0** a = **ST0** (State -> (a, State))

ST0 instead of **DC** – widely used convention

instance **Monad** ~~**ST**~~ **where ...**

instance **Monad** **ST0** **where ...**

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

The application function **apply0**

data **ST0** a = **DC** (State -> (a, State))

to <u>remove</u> (unwrap) the **dummy constructor**,

the application function **apply0** is defined

apply0 :: **ST0** a -> State -> (a, State)

input output

pattern matching is used

an accessor function

like a **runState** function

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

apply0 and DC

```
data ST0 a = DC (State -> (a, State))
```

```
apply0  :: ST0 a -> State -> (a, State)
           input    output
```

unwrapping function

```
DC :: (State -> (a, State)) -> ST0 a
        input                  output
```

wrapping function

an accessor function
like a **runState** function

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

Unwrapping Data Constructor using (**DC g**)

data ST0 a = **DC** (State -> (a, State))   Data Constructor

apply0 :: **ST0** a -> State -> (a, State)   Application Function

apply0 (**DC g**) :: State -> (b, State)   **pattern matching**

apply0 (**DC g**) = **g**

apply0 (**DC g**) s = **g** s

s :: State

g :: State -> (a, State)

g s :: (a, State)

(.) :: (b->c) -> (a->b) -> (a->c)

f . g = \x -> f (g x)

f . g x = f (g x)

~~(**DC** . **f**) x = **DC** (**f** x)~~

not a composite function

but a function argument

(**DC g**) :: DC (State -> (b, State))

(**DC g**) :: ST0 a

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

ST a and ST0 a

No data constructor

```
type ST a = State -> (a, State)

st :: State -> (a, State)
st = \s -> (s, s+1)

st s ::  (a, State)

f    ::  a -> ST a

f  x :: State -> (b, State)
f  x s :: (b, State)
```

With a data constructor : DC

```
data ST0 a = DC (State -> (a, State))

          st0 ::  DC (State -> (a, State))
          st0 =  DC (\s -> (s, s+1))

apply0 st0 ::  State -> (a, State)
apply0 st0 s ::  (a, State)

          f    ::  a -> ST0 a
          f  x ::  ST0 a
          f  x ::  DC (State -> (a, State))
apply0 f  x ::  State -> (a, State)
apply0 f  x s ::  (b, State)
```

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

ST a and ST0 a Examples

t.hs

```
type ST a   =        Int -> (a, Int)
data ST0 a = DC (Int -> (a, Int))

st :: ST Int
st = (\s -> (s, s+1))

st0 :: ST0 Int
st0 = DC (\s -> (s, s+1))

apply0 :: ST0 a -> Int -> (a, Int)
apply0 (DC f)  = f
```

```
:load t.hs
 …
*Main> :t st
st :: ST Int
*Main> :t st0
st0 :: ST0 Int
*Main> :t st 3
st 3 :: (Int, Int)
*Main> :t apply0 st0 3
apply0 st0 3 :: (Int, Int)
*Main>
```

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

apply0 st0 s and apply0 f x s’

```
data ST0 a = DC (State -> (a, State))

apply0  :: ST0 a -> State -> (a, State)
apply0 (DC f) x = f x
```

```
apply0 st0 s      =  (x,s')           s  →  (x,s')
apply0 f x s’     =  (y,s')           s'  →  (y,s')
```

$$\begin{tikzcd} (–,s) \arrow[r] & (x,s’) \arrow[r] & (y,s’) \end{tikzcd}$$

apply0 st0 s ⇨ (x,s')   apply0 f x s' ⇨ (y,s')

```
st0 :: ST0 a
st0 :: DC (State -> (a, State))
st0 = DC (\s -> (s, s+1))
apply0 st0 s ::  (a, State)

f :: a -> ST0 a
f :: a -> DC (State -> (b, State))
f  x :: DC (State -> (b, State))
apply0 f  x s’ :: (b, State)
```

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

st0 >> f using apply0

```
                              (1)    (2)
st >>= f  =  \s -> let (x,s') = st s in f x s'
```

```
type ST a = State -> (a, State)
```

```
                                          (1)             (2)
st0 >>= f  = DC (  \s -> let (x, s') = apply0 st s in apply0 f  x  s'  )
```

```
data ST0 a = DC (State -> (a, State))
```

binding

variables

st s ➡ (x,s')   s → (x,s')

f x s' ➡ (y,s')   s' → (y,s')

apply0 st0 s ➡ (x,s')   s → (x,s')

apply0 f x s ➡ (y,s')   s' → (y,s')

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

ST0 and ST Monad Instance

```
instance Monad ST0 where
  -- return :: a -> ST0 a
 return x   = DC ( \s -> (x,s) )

  -- (>>=)  :: ST0 a -> (a -> ST0 b) -> ST0 b
  st >>= f   = DC(   \s -> let (x, s') = apply0 st s in apply0 (f x) s'   )
```

```
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'
```

the runtime overhead of manipulating the dummy constructor **DC** can be eliminated by defining **ST0** using the **newtype** mechanism

efficiency – enable pointers

```
data        ST0 a = DC (State -> (a, State))
newtype     ST0 a = DC (State -> (a, State))
```

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

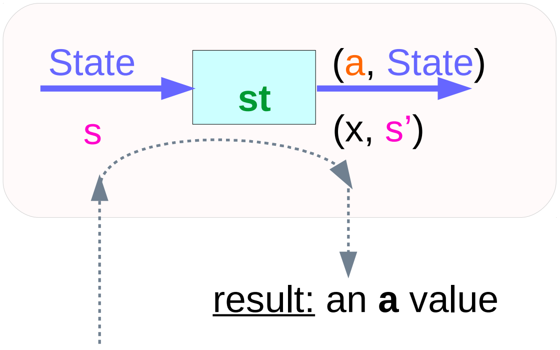
A value of type **ST0 a**

a **value** of type **ST a** (or **ST0 a**) is simply

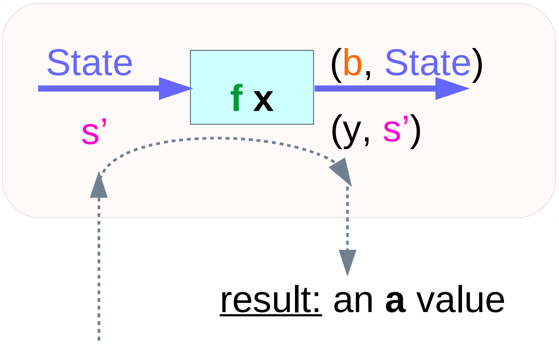
an action that returns an **a** value.

(like state processor function of **State** Monad)

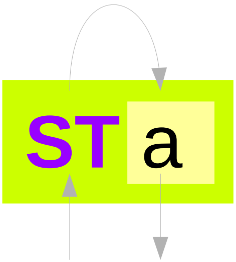
st :: **ST** a



f x :: **ST** a



action



function is a value

State -> (a, State)

function is executable

– taking the inputs

– giving its output

– taking s giving (x, s')

– taking s' giving (y, s')

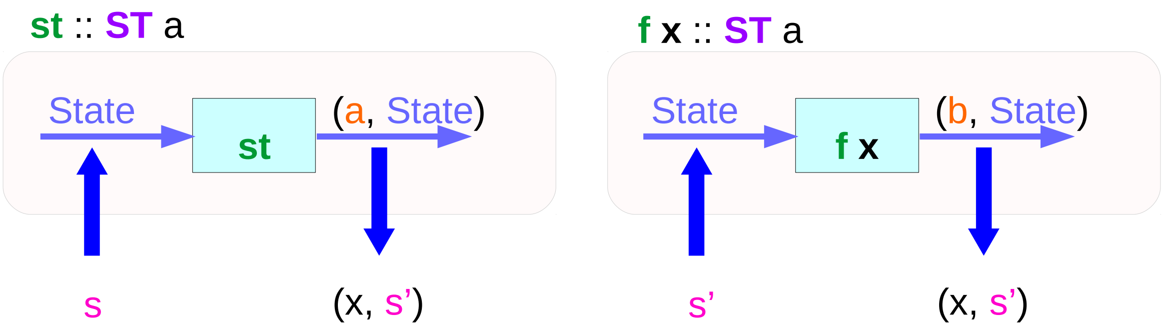
https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

Executing a value of type **ST0 a**

the **apply0** allows us

to **execute** an action from some initial state.

(like **runState** accessor function of **State** Monad)



apply0 st s → (x, s')

apply0 f x s → (x, s')

apply0 function input → output

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

Sequencing Combinator (>>)

The **sequencing combinators** (>>) allow us

to combine simple actions to get bigger actions,

monad

(>>) :: Monad m => m a -> m b -> m b;

(>>=) :: Monad m => m a -> (a -> m b) -> m b;

monad returning function

a1 >> a2 takes the actions **a1** and **a2** and

returns the mega action which is

a1-then-**a2**-returning-the-value-returned-by-**a2**.

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

Sequencer (>>=) and return

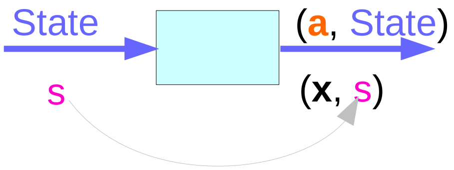
the >>= sequencer is kind of like >>

(>>=) :: Monad m => m a -> (a -> m b) -> m b;

only it allows you to "remember" intermediate values

that may have been returned.

return :: a -> ST0 a



takes a value **x** and yields an action

that doesn't actually change the state,

but just returns the same value **x**

remember x

intermediate

return

action

the same state

a function is a value

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

Do Notation Example

```
pairs :: [a] -> [b] -> [(a,b)]
pairs xs ys =  do x <- xs
                  y <- ys
                  return (x, y)
```

do method

this function returns *all* possible ways

of pairing elements from two lists

each possible value x from the list **xs** x <- **xs**

each possible value y from the list **ys** y <- **ys**

return the pair (x, y).

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

Comprehension Notation Example

```
pairs :: [a] -> [b] -> [(a,b)]
pairs xs ys =  do x <- xs
                  y <- ys
                  return (x, y)
```

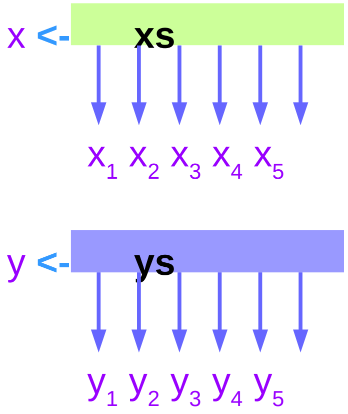
do method

```
pairs xs ys = [(x, y) | x <- xs, y <- ys]
```

comprehension notation

In fact, there is a formal connection between the **do** notation and the **comprehension** notation.

simply different shorthands for repeated use of the **>>=** operator for lists.



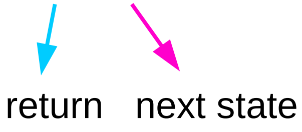
Generators

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

Counter Example (1)

the **state processing function** can be defined
using the notion of a state transformer,
in which the internal state is simply the next fresh integer

```
type State = Int

fresh :: ST0 Int
fresh =  DC (\n -> (n, n+1))
```



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

Counter Example (2)

```
type State = Int

fresh :: ST0 Int
fresh =  DC (\n -> (n, n+1))
```

In order to generate a **fresh** integer,
we define a special state transformer
that simply returns the **current state** as its **result**,
and the **next integer** as the **new state**:

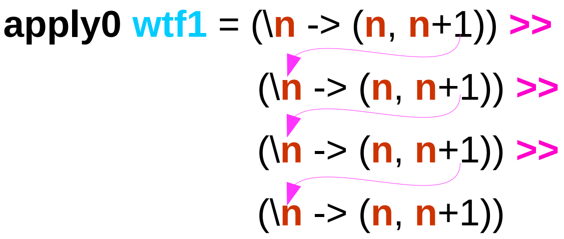
Note that **fresh** is a state transformer
(where the State is itself just **Int**),
that is an action that happens to **return** integer values.

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

Executing **wtf1** (1)

```
type State = Int

fresh :: ST0 Int
fresh =  DC (\n -> (n, n+1))

wtf1 = fresh >>
         fresh >>
         fresh >>
         fresh

ghci> apply0 wtf1 0
```

```
wtf1 = DC (\n -> (n, n+1)) >>
       DC (\n -> (n, n+1)) >>
       DC (\n -> (n, n+1)) >>
       DC (\n -> (n, n+1))
```



```
apply0 wtf1 = (\n -> (n, n+1)) >>
                (\n -> (n, n+1)) >>
                (\n -> (n, n+1)) >>
                (\n -> (n, n+1))
```

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

Executing **wtf1** (2) – executing a fresh

```
data ST0 a = DC (State -> (a, State))
data ST0 a = DC (Int -> (a, Int))
data ST0 Int = DC (Int -> (Int, Int))

apply0  :: ST0 a -> State -> (a, State)
apply0  :: ST0 a -> Int -> (a, Int)
apply0  :: ST0 Int -> Int -> (Int, Int)
apply0     fresh     0  ➡ ( 0 ,  1)
```

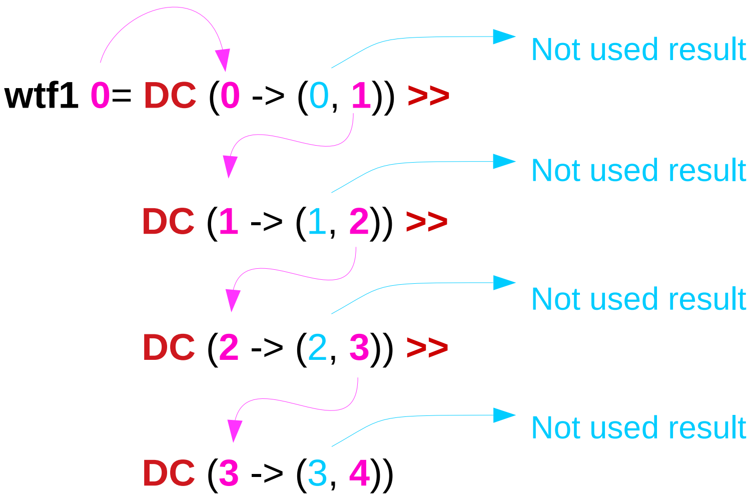
apply0 fresh 0 ➡ (0 , 1)

```
fresh :: ST0 Int
fresh =  DC (\n -> (n, n+1))

apply0 st s =  (x,s')     s → (x,s')
apply0 f x s =  (y,s')    s' → (y,s')
```

```
data ST0 a = DC (Int -> (a, Int))
apply0  :: ST0 a -> Int -> (a, Int)
```

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

Executing **wtf1** (3) – result is not used, state is updated



Not used result

wtf1 0= **DC** (**0** -> (0, **1**)) **>>**

Not used result

DC (**1** -> (1, **2**)) **>>**

Not used result

DC (**2** -> (2, **3**)) **>>**

Not used result

DC (**3** -> (3, **4**))

internal state s

external output x

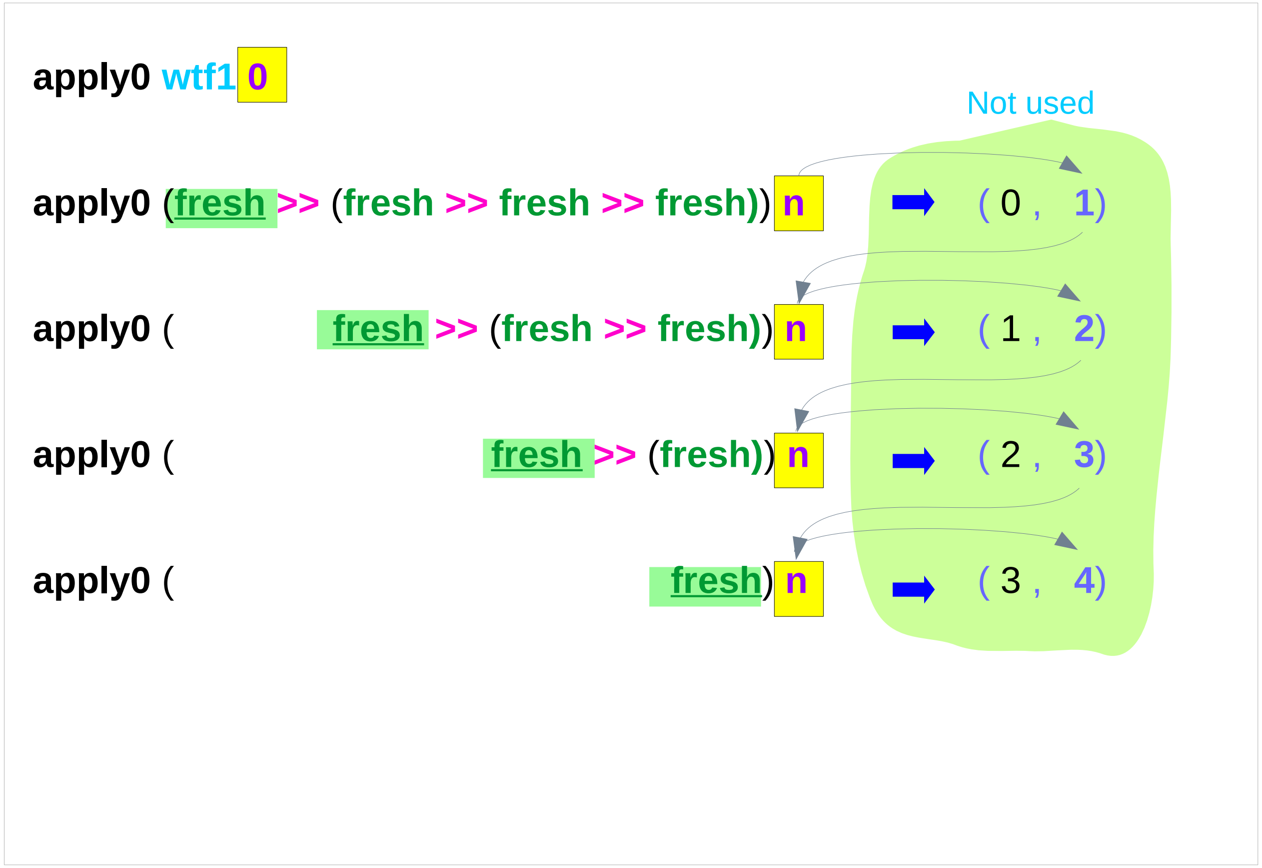
wtf1 0= **DC** (**0** -> (0, **1**)) **>>**
DC (**1** -> (1, **2**)) **>>**
DC (**2** -> (2, **3**)) **>>**
DC (**3** -> (3, **4**))

apply0 wtf1 0 =
(0 -> (0, 1)) **>>**
(1 -> (1, 2)) **>>**
(2 -> (2, 3)) **>>**
(3 -> (3, 4))

n=1
n=2
n=3
n=4
(3,4)

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

Executing **wtf1** (4) – input parameter is updated



apply0 wtf1 0

Not used

apply0 (fresh >> (fresh >> fresh >> fresh)) n ➡ (0 , **1**)

apply0 (fresh >> (fresh >> fresh)) n ➡ (1 , **2**)

apply0 (fresh >> (fresh)) n ➡ (2 , **3**)

apply0 (fresh) n ➡ (3 , **4**)

n = 0 ➡ **n = 1**

n = 1 ➡ **n = 2**

n = 2 ➡ **n = 3**

n = 3 ➡ **n = 4**

```
wtf1 = fresh >>
       fresh >>
       fresh >>
       fresh
```

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

Executing wtf1 (5) – equivalent expressions

```
type State = Int

fresh :: ST0 Int
fresh                                   = DC (\n -> (n+0, n+1))
fresh >> fresh                          = DC (\n -> (n+1, n+2))
fresh >> fresh >> fresh                 = DC (\n -> (n+2, n+3))
fresh >> fresh >> fresh >> fresh        = DC (\n -> (n+3, n+4))
```

```
wtf1 = fresh >>
       fresh >>
       fresh >>
       fresh
```

```
wtf1 = DC (\n -> (n+3, n+4))
```

```
wtf1 = DC (\n -> (n, n+1)) >>
       DC (\n -> (n, n+1)) >>
       DC (\n -> (n, n+1)) >>
       DC (\n -> (n, n+1))
```

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

Executing wtf2

```
wtf2 = fresh >>= \n1 ->          n1 = 0      intermediate result
       fresh >>= \n2 ->          n2 = 1      intermediate result
       fresh >>
       fresh >>                    f1: monad returning function
       return [n1, n2]             f2: monad returning function
```

```
wtf2 = fresh >>=
                 intermediate result
       ( \n1 -> fresh >>=
                        intermediate result
             (\n2 ->  fresh >> fresh >>  return [n1, n2])  )
```

```
*Main> apply0 wtf2 0
([0,1],4)
```

```
apply0 wtf2 0 =
  (0 -> (0, 1)) >>=\n1 ->        n=1, n1=0
  (1 -> (1, 2)) >>=\n2 ->        n=2, n2=1
  (2 -> (2, 3)) >>               n=3
  (3 -> (3, 4)) >>               n=4
  return [n1, n2]                ([0,1], 4)
```

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

Executing wtf2‘

```
wtf2' = do  { n1 <- fresh;                    n1 = 0
              n2 <- fresh;                    n2 = 1
              fresh ;
              fresh ;
              return [n1, n2];
            }
```

```
      do {      ;      ;      }      semicolon necessary
```

```
*Main> apply0 wtf2' 0
([0,1],4)
```

```
wtf2 = fresh >>= \n1 ->
       fresh >>= \n2 ->
       fresh >>
       fresh >>
       return [n1, n2]
```

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

Executing wtf3

```
wtf3 = do  n1 <- fresh                       n1=0
           fresh
           fresh
           fresh
           return n1                         3 → (0, 4)   instead of (3, 4)

*Main> apply0 wtf3 0
(0,4)
```

```
apply0 wtf3 0 =
  (0 -> (0, 1)) >>=\n1 ->        n=1, n1=0
  (1 -> (1, 2)) >>=\n2 ->        n=2
  (2 -> (2, 3)) >>               n=3
  (3 -> (3, 4)) >>               n=4
  return [n1, n2]                (0, 4)
```

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

Executing wtf4

```
wtf4 = fresh >>= \n1 ->              n1 = 0
          fresh >>= \n2 ->           n2 = 1
          fresh >>= \n3 ->           n3 = 2
          fresh >>
          return (n1+n2+n3)

*Main> apply0 wtf4 0
(3,4)
```

```
apply0 wtf4 0 =
  (0 -> (0, 1)) >>=\n1 ->      n=1, n1=0
  (1 -> (1, 2)) >>=\n2 ->      n=2, n2=1
  (2 -> (2, 3)) >>=\n3 ->      n=2, n3=2
  (3 -> (3, 4)) >>             n=4
  return (n1+n2+3)             (0+1+2, 4)
```

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

Make Functor and Applicative Instances

```
import Control.Applicative
import Control.Monad (liftM, ap)


instance Functor ST0 where
  fmap = liftM


instance Applicative ST0 where
  pure  = return
  (<*>) = ap
```

```
newtype ST0 a = DC (Int -> (a, Int))


instance Monad ST0 where
  return x   = DC( \s -> (x,s) )
  st >>= f   = DC( \s -> let (x, s') = apply0 st s
                         in apply0 (f x) s' )
```

https://stackoverflow.com/questions/31652475/defining-a-new-monad-in-haskell-raises-no-instance-for-applicative

Example Code Listing

```
apply0 :: ST0 a -> Int -> (a, Int)
apply0 (DC f)   = f

fresh :: ST0 Int
fresh = DC (\n -> (n, n+1))

wtf1 = fresh >>
          fresh >>
          fresh >>
          fresh

wtf2 =  fresh >>= \n1 ->
           fresh >>= \n2 ->
           fresh >>
           fresh >>
           return [n1, n2]
```

```
wtf2' = do {  n1 <- fresh ;
                    n2 <- fresh ;
                    fresh ;
                    fresh ;
                    return [n1, n2] ;
                 }

wtf3 = do   n1 <- fresh
                 fresh
                 fresh
                 fresh
                 return n1

wtf4 = fresh >>= \n1 ->
          fresh >>= \n2 ->
          fresh >>= \n3 ->
          fresh (n1+n2+n3)
```

Results

```
*Main> :load st.hs
[1 of 1] Compiling Main           ( st.hs, interpreted )
Ok, modules loaded: Main.



*Main> apply0 (fresh) 0
(0,1)
*Main> apply0 (fresh >> fresh) 0
(1,2)
*Main> apply0 (fresh >> fresh >> fresh) 0
(2,3)
*Main> apply0 (fresh >> fresh >> fresh >> fresh) 0
(3,4)
```

```
*Main> apply0 wtf1 0
(3,4)

*Main> apply0 wtf2 0
([0,1],4)

*Main> apply0 wtf2' 0

([0,1],4)


*Main> apply0 wtf3 0

(0,4)


*Main> apply0 wtf4 0

(3,4)
```

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

Transformer Stacks

making a double, triple, quadruple, ... monad
by wrapping around existing monads
that provide wanted functionality.

You have an innermost monad (usually Identity or IO
but you can use any monad). You then wrap monad transformers
around this monad to make bigger, better monads.

a ➡ M a ➡ N M a ➡ O N M a

To do stuff in an inner monad → cumbersome → monad transformers

lift $ lift $ lift $ foo

https://wiki.haskell.org/Monad_Transformers_Explained

References

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

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