Monad P3 : ST Monad Basics (3A)

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

Transforms a state and returns a value

data ST s a the strict state-transformer monad

A **computation** of type **ST s a**

transforms an **internal state** indexed by **s**

returns a **value** of type **a**.

updated state **s**

returned result type **a**

http://hackage.haskell.org/package/base-4.11.1.0/docs/Control-Monad-ST.html

Mutable Reference Types

data ST s a

For **mutability**,

Data.STRef provides STRefs**.**

Data.Array.ST provides STArrays and **STUArray**s**.**

these allow programmers to produce **imperative code**

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

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

Imperative code is enabled

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

mutable variable of the type **STRef s a** enables the followings

- a **variable** is directly updated,
  rather than a **new value** is formed and
  passed to the **next iteration** of the function.
- **memory modification in place** is also possible

while maintaining the **purity** of a function by using **runST**

functions written using the **ST monad**
appear completely **pure** to the rest of the program.

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

Side effects confined

data ST s a

ST monad code can have internal **side effects**

- destructively updating **mutable variables** and **arrays**,
- confining these **effects** inside the monad.

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

Mutable Reference Type

data STRef s a

mutable references in the (strict) **ST** monad.

a **value** of type **STRef s a** is

a **mutable variable** in **state thread s**,

containing a **value** of type **a**

https://hackage.haskell.org/package/base-4.9.0.0/docs/Data-STRef.html

In place modification

STRef s a mutable reference enables

in place modifications of the variable **n**

- possible by using the type **STRef s a**
- would be considered asa **side effect**
- carried out in a safe and deterministic way

  while preserving the **functional purity**

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

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

Imperative code example – **sumST**

Imperative style code example

that takes a **list** of **numbers**, and **sums** them,

using a **mutable variable**:

a version of the function ***sum*** is defined,

in a way that **imperative languages** are used

taken from the Haskell wiki page on the **ST monad**

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

sumST example – imperative style

```
import Control.Monad.ST
import Data.STRef
import Data.Foldable

sumST :: Num a => [a] -> a
sumST xs = runST $ do
    n <- newSTRef 0
    for_ xs $ \x ->
        modifySTRef n (+x)
    readSTRef n
```

Imperative style code to sum elements of a list

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

sum example – functional style

```
sum :: [a] -> a
sum []     = 0
sum (x:xs) = x + sum xs

product :: [a] -> a
product []     = 1
product (x:xs) = x * product xs

concat :: [[a]] -> [a]
concat []     = []
concat (x:xs) = x ++ concat xs
```

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

s parameter

data ST s a the strict state-transformer monad

the **s parameter** keeps the **internal states**
of different **invocations** of **runST** separate
from other invocations of **runST** and
from invocations of **stToIO**.

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

stToIO :: ST RealWorld a -> IO a

http://hackage.haskell.org/package/base-4.11.1.0/docs/Control-Monad-ST.html

s parameter instance

data ST s a the strict state-transformer monad

the s **parameter** is

an uninstantiated **type variable**

(inside invocations of **runST**),

RealWorld

(inside invocations of **stToIO**).

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

stToIO :: ST RealWorld a -> IO a

http://hackage.haskell.org/package/base-4.11.1.0/docs/Control-Monad-ST.html

runST method

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

return the **value a** computed

by a **state transformer computation**. ……. **ST s a**

The **forall** ensures that

the **internal state s** used by the **ST** computation ……. **ST s a**

is inaccessible to the rest of the program.

http://hackage.haskell.org/package/base-4.11.1.0/docs/Control-Monad-ST.html

runST method extracts a value

There is one major difference

that sets apart **ST** from both **State** and **IO**.

***runST* extracts a value**

Control.Monad.ST offers a **runST** function

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

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

runState vs runST

to get out of the State monad,

use runState ………. s -> (s, a) …….. function

to get out of the ST monad,

use runST …………….. a ..……………. value

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

```
newtype ST s a = ST (State# s -> (# State# s, a #))
runST :: forall a. (forall s. ST s a) -> a
```

runState :: s -> (s, a)
State Constructors

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

forall s. in an argument

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

non conventional monad method type signature

extract **a** values from the **ST monad value**

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

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

Uninstantiated **s** value

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

- tells the **type checker** **s** could be anything.

- *do not make any assumptions about it.*

→ this means **s** *cannot be matched with anything*
even with the **s** *from another invocation of* ***runST***

uninstantiated s value

an **existential type**
the only thing we know about it is that it exists.

Uninstantiated value

Existential type s

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

Existential type **s**

The **s** makes the type system prevent you

from doing things which would be **unsafe**.

It doesn't "do" anything at run-time;

it just makes the **type checker**

reject programs that do dubious things.

(It is a so-called **phantom type**,

a thing with only **exists**

in the type checker's view,

and doesn't affect anything at **run-time**.)

https://stackoverflow.com/questions/12468622/how-does-the-st-monad-work

Local state **s** in an ST action

By using an ***uninstantiated*** **s** **value**,
we can ensure that we aren't "cheating"
and running arbitrary **IO actions** inside an **ST action**.

Instead, we just have "**local state**" ***modifications***,
for ***some definition*** of **local state**.

the details of using **ST** correctly and
the **Rank2Types** approach to **runST**

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

Local State
Thread Safety
Compartmentalize
ST Escape Mechanism

https://haskell-lang.org/tutorial/primitive-haskell

ST monad thread and reference

The **ST monad** lets you use update-in-place,
but is escapable (unlike **IO**).

ST actions have the form:

ST s a

return a **value** of type **a**
execute in **thread s**.

all **reference** types are tagged with the **thread s**,
so that **actions** (**ST s a**) can only affect
references in their own **thread** (**s**)

reference type
forall. s

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

Thread safety constraint

a **mutable reference** created in <u>one</u> **ST computation**,

<u>cannot</u> be used in <u>another</u> **ST computation**

We don't want to allow this because of **thread-safety**

ST computations are <u>not</u> allowed to assume that

the **<u>initial</u> internal environment**

contains **any <u>specific</u> references**.

Local State

Thread Safety

Compartmentalize

ST Escape Mechanism

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

Compartmentalize

The key feature of the **existential** is that
it allows the compiler to **generalize**
the **type** of the **state** in the first parameter,
and so the **result type** cannot depend on it.

This neatly sidesteps our dependence problems,
and '**compartmentalizes**' each call to **runST** into its own little heap,
with **references** not being able to be shared between different calls.

creating a **reference** in one **ST computation**,
It cannot be used in another **ST computation**

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

X

Local State
Thread Safety
Compartmentalize
ST Escape Mechanism

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

Escaping an ST action

the type of the function used to escape **ST** is:

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

The **action** you pass must be (**ST s a**)

universal in **s** (**forall s. ST s a**)

so inside your **action** you don't know what **thread** (**s**),
thus you cannot access any other **threads**,
thus **runST** is **pure**.

uninstantiated s

Local State
Thread Safety
Compartmentalize
ST Escape Mechanism

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

ST escape mechanism

The **ST** **monad** also provides **mutable state**,

but it does have an **escape mechanism**

— the **runST** function.

This lets you convert an **impure value** into a **pure** one.

https://stackoverflow.com/questions/28769550/what-is-the-difference-between-iotost-and-unsafeiotost-from-ghc-io

ST escape mechanism – safety measures

But now it is impossible to guarantee

what **order** separate ST **blocks** will run in.

(uninstantiated s of an existential type)

But it is possible to ensure that

separate ST **blocks** can't "interfere" with each other.

You can access **mutable state**,

but that **state** cannot escape the **ST block**.

For that reason, you cannot perform

any I/O operations in the ST **monad**.

https://stackoverflow.com/questions/28769550/what-is-the-difference-between-iotost-and-unsafeiotost-from-ghc-io

In-place quicksort

runST is **pure**.

this is very useful, since it allows you to implement
externally pure things like in-place quicksort,
and present them as **pure** functions

∀ e. Ord e ⇒ Array e -> Array e;

without using any **unsafe** functions.

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

ST vs. **IO** Monad – internal state

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

IO is **isomorphic** to **ST RealWorld**.

ST works under the *exact same* principles as **IO**

mutable references in the **ST monad**

are possible through **threading state**

https://haskell-lang.org/tutorial/primitive-haskell

ST vs. **IO** Monad – the special internal state **Realworld**

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

The RealWorld parameter indicates
that the **internal state** used by the ST **computation** s
is a special one supplied by the IO **monad**, RealWorld
and thus distinct from those used by invocations of **runST**.

http://hackage.haskell.org/package/base-4.11.1.0/docs/Control-Monad-ST.html
https://haskell-lang.org/tutorial/primitive-haskell

stToIO

Since **ST RealWorld** is isomorphic to **IO**,

we should be able to convert between the two of them.

stToIO :: ST RealWorld a -> IO a

can embed a **strict state transformer** **ST** in an **IO** **action**.

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

https://haskell-lang.org/tutorial/primitive-haskell

Realworld Type

data RealWorld

RealWorld is deeply magical.

It is **primitive**,

but it is not **unlifted** (hence ptrArg).

we never manipulate **values** of type RealWorld;

it's only used in the type system, to parameterise **State#**.

State# RealWorld

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

http://hackage.haskell.org/package/base-4.11.1.0/docs/Control-Monad-ST.html

Lazy evalation

By default, Haskell uses **lazy evaluation**

when you call a **function**,

the body will not *execute* immediately,

The body will only be actually executed

when the **result** of the **function**

is used in an **IO** **computation**,

http://hackage.haskell.org/package/base-4.11.1.0/docs/Control-Monad-ST.html

Strict evaluation

Strict Haskell gives Haskell **strict evaluation**,
which is the kind of evaluation most other languages have,
and hence makes it easier to reason about **performance**.

mtl package provides two types of **State** monad;

Control.Monad.State.Strict
Control.Monad.State.Lazy. …. **Control.Monad.State**

https://www.reddit.com/r/programming/comments/3sux1d/strict_haskell_xstrict_has_landed/
https://kseo.github.io/posts/2016-12-28-lazy-vs-strict-state-monad.html

ST Monad – a strict monad

data ST s a the **strict** state-transformer monad

The **>>=** and **>>** operations are **strict** in the **state** ……. **s**

(though not **strict** in **values** stored in the state). ………… **a**

runST (writeSTRef _|_ v >>= f) = _|_

http://hackage.haskell.org/package/base-4.11.1.0/docs/Control-Monad-ST.html

ST Monad – a strict monad

```
runST (writeSTRef _|_ v >>= f) = _|_

writeSTRef :: STRef s a -> a -> ST s ()
writeSTRef s v :: ST s ()

(>>=) :: ST s a -> (a -> ST s b) -> ST s b
f :: a -> ST s b
(writeSTRef s v >>= f) :: ST s b

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

http://hackage.haskell.org/package/base-4.11.1.0/docs/Control-Monad-ST.html

fixST method

fixST :: (a -> ST s a) -> ST s a

allow the **result** of a **state transformer computation**

to be used (lazily) inside the **computation**.

Note that if **f** is **strict**, **fixST f = _|_**.

http://hackage.haskell.org/package/base-4.11.1.0/docs/Control-Monad-ST.html

Case examples

```
Input: case 2 of { (1) -> "A"; (2) -> "B"; (3) -> "C" }
Output: "B"
```

```
aaa x = case x of
        1 ->  "A"
        2 ->  "B"
        3 ->  "C"
Input: aaa 3
Output: "C"
```

```
aaa x = case x of
        []      -> [1]
        [x]     -> [x]
        (x:xs) -> xs

Input: aaa [1,2,3]
Output: [2,3]

Input: aaa []
Output: [1]

Input: aaa [4]
Output: [4]
```

https://stackoverflow.com/questions/12468622/how-does-the-st-monad-work

ST s a Type Definition

```
newtype ST s a = ST (STRep s a)

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

https://stackoverflow.com/questions/12468622/how-does-the-st-monad-work

(ST s) Monad

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

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

(ST s) Monad

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

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

```
(m s)  ➡  (# new_s, r #)
(ST m) >>= k
ST m :: ST s a
m :: STRep s a
m :: State# s -> (# State# s, a #)
(m s) :: (# State# s, a #)
(# new_s, r #) :: (# State# s, a #)

(k r)  ➡  ST k2
k :: a -> ST s a          r :: a
(k r) :: ST s a
ST k2 :: ST s a
k2 :: STRep s a
k2 :: State# s -> (# State# s, a #)
k2 new_s ::  (# State# s, a #)
```

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

(ST s) Monad

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

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

(m s) ➡ (# new_s, r #)

(ST m) >>= k

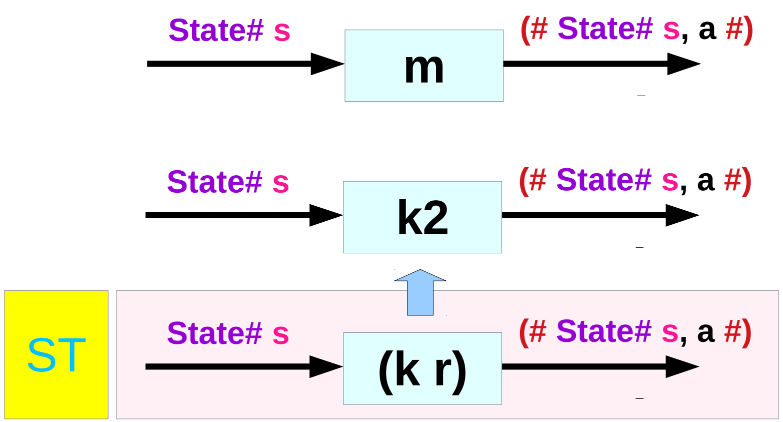
ST m :: ST s a

m :: STRep s a

m :: State# s -> (# State# s, a #)

(m s) :: (# State# s, a #)

(# new_s, r #) :: (# State# s, a #)



m :: STRep s a

k2 :: STRep s a

(k r) :: ST s a

(k r) ➡ ST k2

k :: a -> ST s a   r :: a

(k r) :: ST s a

ST k2 :: ST s a

k2 :: STRep s a

k2 :: State# s -> (# State# s, a #)

k2 new_s :: (# State# s, a #)

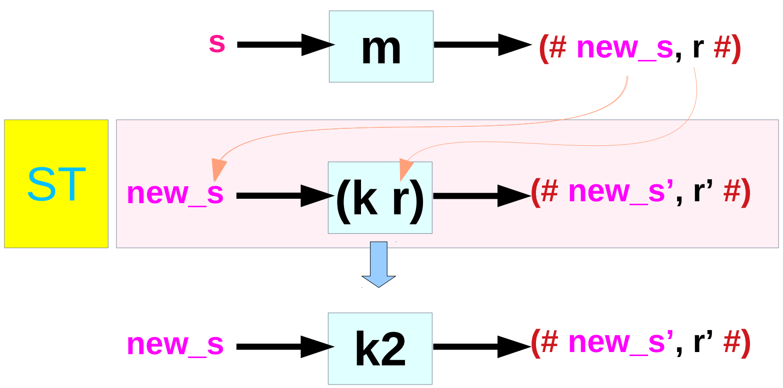
http://hackage.haskell.org/package/base-4.12.0.0/docs/Control-Monad-ST.html

(ST s) Monad

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

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

```
(m s) ➡ (# new_s, r #)
(ST m) >>= k
ST m :: ST s a
m :: STRep s a
m :: State# s -> (# State# s, a #)
(m s) :: (# State# s, a #)
(# new_s, r #) :: (# State# s, a #)
```




```
m :: STRep s a
(k r) :: ST s a
k2 :: STRep s a
```

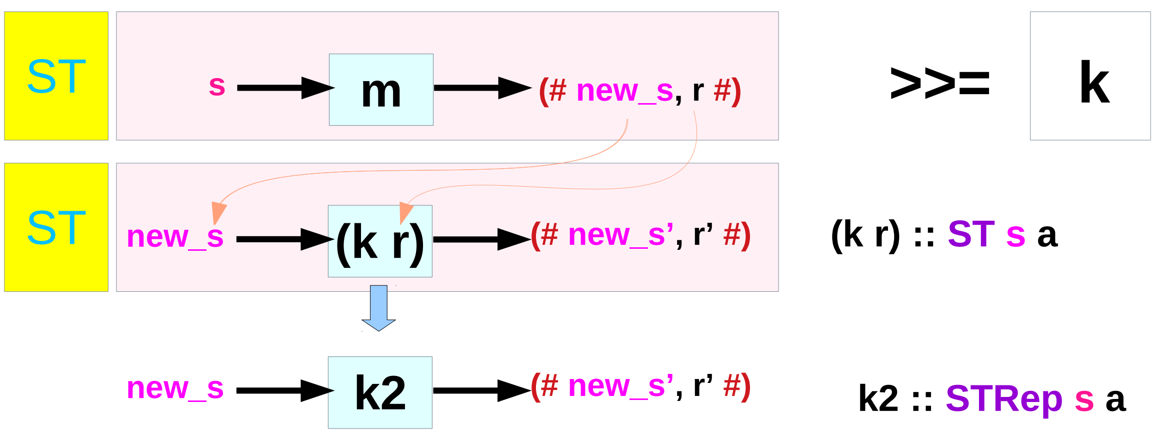
```
(k r) ➡ ST k2
k :: a -> ST s a          r :: a
(k r) :: ST s a
ST k2 :: ST s a
k2 :: STRep s a
k2 :: State# s -> (# State# s, a #)
k2 new_s :: (# State# s, a #)
```

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

(ST s) Monad

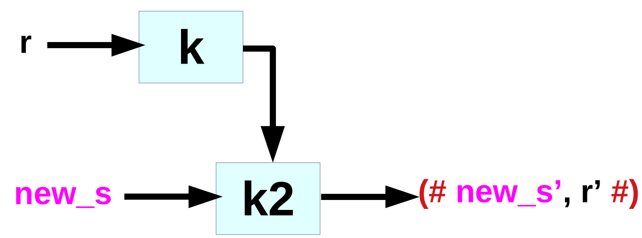
```
newtype ST s a = ST (STRep s a)
type STRep s a = State# s -> (# State# s, a #)


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



(k r) :: ST s a

k2 :: STRep s a



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

IO Monad

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

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


       (IO m) >>= k
       IO m :: IO a          m :: State# RealWorld -> (# State# RealWorld, a #)
       k :: a -> IO b        k a :: IO b


         s  :: State# RealWorld
         s' :: State# RealWorld
       m s :: (# State# RealWorld, a #)
 (# s', a #) :: (# State# RealWorld, a #)
```

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

IO Monad

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

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

unIO :: IO a -> (State# RealWorld -> (# State# RealWorld, a #))
unIO (IO a) = a

     k :: a -> IO b      k a :: IO b

           unIO (k a) :: State# RealWorld -> (# State# RealWorld, a #)
                s' :: State# RealWorld
          unIO (k a) s' :: (# State# RealWorld, a #)
       \s -> unIO (k a) s' :: State# RealWorld -> (# State# RealWorld, a #)
   IO $ \s -> unIO (k a) s' :: IO b
```

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

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

References

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

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