Monad P3 : ST Monad Methods (3B)

Young Won Lim
10/23/19

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

Thread safety constraint

creating a **reference** in one **ST computation**,

It cannot be used in another **ST computation**

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

no ST computation should be allowed to assume that

the **initial internal environment** contains **any specific references**.

More concretely, we want the following code to be **invalid**:

Example: Bad ST code

```
let v = runST (newSTRef True)
in runST (readSTRef v)
```

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

Thread safety constraint

Example: Bad ST code

```
let v = runST (newSTRef True)
in runST (readSTRef v)
```

```
newSTRef :: a -> ST s (STRef s a)
```

STRef s True

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

if **s** in the <u>first</u> parameter
it <u>cannot</u> in the <u>second</u>
scope constraint

```
readSTRef :: STRef s a -> ST s a
```

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

reference in one **ST computation**
cannot be used
in another **ST computation**
thread safety constrain

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

Scope constraint

The **effect** of the **rank-2 polymorphism** in **runST**'s type
is to constrain the **scope** of the **type variable s**
to be within the first parameter

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

if the type variable **s** appears in the first parameter
it cannot also appear in the second.

Example: Briefer bad ST code

```
... runST (newSTRef True) ...
```

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

Type checking

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

Example: The compiler's type checking stage

```
newSTRef True :: forall s. ST s (STRef s Bool)
(forall s. ST s (STRef s Bool)) -> STRef s Bool
```

The importance of the **forall** in the first bracket is that we can change the name of the **s**.

Example: A type mismatch!

```
(forall s'. ST s' (STRef s' Bool)) -> STRef s Bool
```

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

A type mismatch

giving the variable a different label :

as in mathematics, saying ∀x. x > 5

is precisely the same as saying ∀y. y > 5 ;

because the **forall** does not scope over the **return type** of **runST**,

we don't rename the **s** there as well.

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

But suddenly, we've got a **type mismatch**!

The result type of the **ST** computation in the first parameter

must match the **result** type of **runST**, but now it doesn't't!

(forall s'. ST s' (STRef s' Bool)) -> STRef s Bool

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

The scope of **s** in **forall s.**

let ref = runST $ newSTRef (4 :: Int)

newSTRef :: a -> ST s (STRef s a)

newSTRef Int :: ST s (STRef s Int)

type variable **s** should <u>not</u> escape its scope

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

runST :: (forall s. ST s (STRef s Int)) -> (STRef s Int)

cannot match type **a** with **STRef s a**

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

type variable **s**'s scope

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

Bad ST code 1 – no reference passing

Example: Bad ST code

```
let v = runST (newSTRef True) ……. the 1st ST computation (runST)

in runST (readSTRef v) ……………… the 2nd ST computation (runST)
```

no ST computation should be allowed to assume that
the **initial internal environment** contains **any specific references**.

the above code is **invalid**:

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

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

Bad ST code 1 – no scope escape

```
let v = runST (newSTRef True)
in runST (readSTRef v)

    newSTRef :: a -> ST s (STRef s a)
    newSTRef True :: ST s (STRef s Bool)

    runST :: forall a. (forall s. ST s a) -> a
    runST :: forall a. (forall s. ST s (STRef s Bool)) -> (STRef s Bool)
    runST (newSTRef True) :: STRef s Bool
    v :: STRef s Bool

    readSTRef :: STRef s a -> ST s a
    readSTRef v :: ST s Bool

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

type variable s should not escape its scope

cannot match type a with STRef s a

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

Bad ST code 2

let ref = runST $ newSTRef (4 :: Int)

newSTRef :: a -> ST s (STRef s a)
newSTRef 4 :: ST s (STRef s Int)

type variable **s** should not escape its scope

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

cannot match type **a** with **STRef s a**

runST cannot extract any **reference** of the type **(STRef s a)**

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

Bad ST code 2 – error messages

Attempt to keep an **STRef** around to pass to pure code:

```
GHCi> import Control.Monad.ST
GHCi> import Data.STRef
GHCi> let ref = runST $ newSTRef (4 :: Int)
```

```
<interactive>:125:19:
    Couldn't match type ‘a’ with ‘STRef s Int’
      because type variable ‘s’ would escape its scope
    This (rigid, skolem) type variable is bound by
      a type expected by the context: ST s a
      at <interactive>:125:11-37
    Expected type: ST s a
      Actual type: ST s (STRef s Int)
    Relevant bindings include ref :: a (bound at <interactive>:125:5)
    In the second argument of ‘($)’, namely ‘newSTRef (4 :: Int)’
    In the expression: runST $ newSTRef (4 :: Int)
```

“because type variable s would escape its scope"

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

Read the value of an **STRef**

```
newSTRef (4 :: Int)
        :: ST s (STRef s Int)

runST $ newSTRef (4 :: Int)
        :: STRef s Int
```

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

=<< and >>=

(=<<) :: Monad m => (a -> m b) -> m a -> m b infixr 1

Same as >>=, but with the arguments interchanged.

(>>=) :: forall a b. m a -> (a -> m b) -> m b infixl 1

https://www.stackage.org/haddock/lts-13.27/base-4.12.0.0/Control-Monad.html#v:-62--62--61-

Bad ST code 3 – not the same **s**

```
let x = runST $ readSTRef =<< runST (newSTRef (4 :: Int))
```

1. newSTRef :: a -> ST s (STRef s a)
   newSTRef (4 :: Int) :: ST s (STRef s Int)

2. runST :: (forall s. ST s a) -> a
   ST s (STRef s Int)) -> (STRef s Int)
   runST(newSTRef (4 :: Int)) :: (STRef s Int)

3. readSTRef :: STRef s a -> ST s a
   runST :: (forall s. ST s a) -> a
   readSTRef =<< runST (newSTRef (4 :: Int)) :: (ST s Int)

4. runST :: (forall s. ST s a) -> a
   (runST $ readSTRef =<< runST (newSTRef (4 :: Int))) :: Int

The 's' from each computation are necessarily not the same.

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

Read the value of an STRef

```
newSTRef (4 :: Int)
      :: ST s (STRef s Int)

runST $ newSTRef (4 :: Int)
      :: STRef s Int
```

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

Bad ST code 3

let x = runST $ readSTRef =<< runST (newSTRef (4 :: Int))

1. **newSTRef :: a -> ST s (STRef s a)**
newSTRef (4 :: Int) :: ST s (STRef s Int)

2. **runST :: (forall s. ST s a) -> a**
ST s1 (ST s (STRef s Int)) -> (ST s (STRef s Int))
runST(newSTRef (4 :: Int)) :: (ST s (STRef s Int))

3. **readSTRef :: STRef s a -> ST s a**
STRef s1 Int -> ST s1 Int

Couldn't match type '**STRef s1 Int**' with '**ST s (STRef s a)**'

Expected type: **ST s1 (ST s (STRef s a))**
Actual type: **ST s1 (STRef s1 Int)**

The '**s**' from each computation are necessarily not the same.

runST :: (forall s. ST s a) -> a
newSTRef :: a -> ST s (STRef s a)
readSTRef :: STRef s a -> ST s a

Read the value of an **STRef**

newSTRef (4 :: Int)
:: ST s (STRef s Int)

runST $ newSTRef (4 :: Int)
:: STRef s Int

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

Bad ST code 3 – error messages

Attempt to feed an STRef from one ST computation to another:

```
GHCi> import Control.Monad.ST
GHCi> import Data.STRef
GHCi> let x = runST $ readSTRef =<< runST (newSTRef (4 :: Int))

<interactive>:129:38:
    Couldn't match type ‘STRef s1 Int’ with ‘ST s (STRef s a)’
    Expected type: ST s1 (ST s (STRef s a))
      Actual type: ST s1 (STRef s1 Int)
    Relevant bindings include x :: a (bound at <interactive>:129:5)
    In the first argument of ‘runST’, namely ‘(newSTRef (4 :: Int))’
    In the second argument of ‘(=<<)’, namely
      ‘runST (newSTRef (4 :: Int))’
```

The 's' from <u>each computation</u> are necessarily <u>not the same</u>.

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

Read the value of an STRef

```
newSTRef (4 :: Int)
        :: ST s (STRef s Int)

runST $ newSTRef (4 :: Int)
        :: STRef s Int
```

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

Bad ST Code 4

The s keeps objects inside the ST **monad**

from leaking to the outside of the ST **monad**.

```
-- This is an error... but let's pretend for a moment...
let a = runST $ newSTRef (15 :: Int)
    b = runST $ writeSTRef a 20
    c = runST $ readSTRef a
in b `seq` c
```

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

Bad ST Code 4

```
let a = runST $ newSTRef (15 :: Int)

    b = runST $ writeSTRef a 20

    c = runST $ readSTRef a

in b `seq` c
```

this is a type error

we don't want **STRef** to leak outside the original computation!

the extra **s** in **runST** causes a type error

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

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

Bad ST Code 4

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

the **s** on the computation that you're performing

has to have no **constraints** on it.

So when you try to evaluate a

a = runST (newSTRef (15 :: Int) :: forall s. ST s (STRef s Int))

a :: STRef s Int,

this is wrong since the **s** has "escaped"

outside of the **forall** in **runST**.

runST :: (forall s. ST s a) -> a
newSTRef :: a -> ST s (STRef s a)
readSTRef :: STRef s a -> ST s a

Read the value of an **STRef**

newSTRef (4 :: Int)
:: ST s (STRef s Int)

runST $ newSTRef (4 :: Int)
:: STRef s Int

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

Bad ST Code 4

```
runST :: (forall s . ST s a) -> a
a = runST (newSTRef (15 :: Int) :: forall s. ST s (STRef s Int))
a :: STRef s Int,
```

this is wrong since the s has "escaped"

outside of the **forall** in **runST**.

type variables (e.g. **s**) always have to

appear on the inside of a **forall**,

Haskell allows implicit **forall** quantifiers everywhere.

There's simply no rule that allows you to to meaningfully

figure out the return type of a.

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

Read the value of an **STRef**

```
newSTRef (4 :: Int)
    :: ST s (STRef s Int)

runST $ newSTRef (4 :: Int)
    :: STRef s Int
```

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

Bad ST Code 4

Another example with forall:

To clearly show why you can't allow things to escape a forall,

here is a simpler example:

```
f :: (forall a. [a] -> b) -> Bool -> b
f g flag =
  if flag
  then g "abcd"
  else g [1,2]

> :t f length
f length :: Bool -> Int


> :t f id
-- error --
```

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

Code for both IO and ST

The **IO monad** and the **ST monad** are actually the same monad.

And an **IORef** is actually an **STRef**, and so on.

So it would not so be useful to be able to write code

and use it in both monads.

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

s in ST & STRef

the **phantom s type** in the type signatures.

to run an **ST block**, it needs to work for all possible **s**:

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

All the **mutable stuff** has **s** in the type as well, **STRef s a**

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

to return **mutable stuff** out of the **ST monad**

will be **ill-typed**.

```
data STRef s a = STRef (MutVar# s a)
newtype ST s a = ST (State# s -> (# State# s, a #))
```

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

ST vs. **IO** Monad

```
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

Mutable reference interface

```
newSTRef :: a -> ST s (STRef s a)
readSTRef :: STRef s a -> ST s a
writeSTRef :: STRef s a -> a -> ST s ()

newSTRef init = ST $ \s1# -> (# s2#, STRef var# #)
readSTRef (STRef var#) = ST $ \s2# -> (# State# s3#, val #)
writeSTRef (STRef var#) val = ST $ \s3# ->  (# s4#, () #)

STRef var# :: STRef s a
var# :: MutVar# s a

data STRef s a = STRef (MutVar# s a)
newtype ST s a = ST (State# s -> (# State# s, a #))
```

*** memorization purpose**

mutable references in the **ST monad**

are possible through **threading state**

s1#, s2#, s3#, …

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

STRef Methods

data STRef a A mutable variable in the IO monad

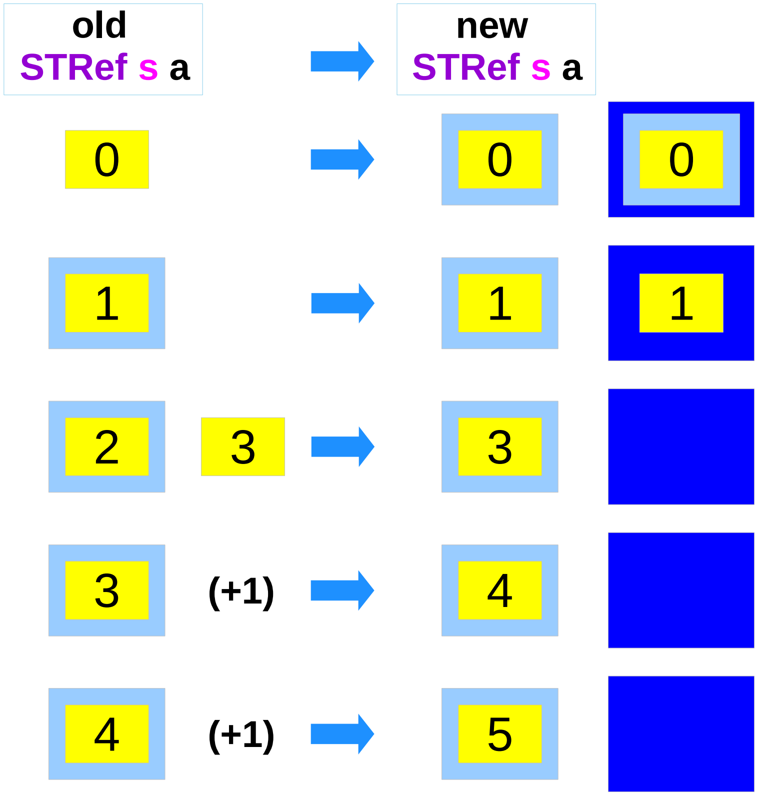
newSTRef :: a -> ST s (STRef s a)
Build a new STRef

readSTRef :: STRef s a -> ST s a
Read the value of an STRef

writeSTRef :: STRef s a -> a -> ST s ()
Write a new value into an STRef

modifySTRef :: STRef s a -> (a -> a) -> ST s ()
Mutate the contents of an STRef.

modifySTRef‘ :: STRef s a -> (a -> a) -> ST s ()
Strict version of modifySTRef



ST s (STRef s a)
ST s a
ST s ()

http://hackage.haskell.org/package/base-4.12.0.0/docs/Data-IORef.html

STRef methods example (1)

data STRef s a

a value of type **STRef s a** is a mutable variable in state thread **s**, containing a value of type **a**

```
>>> :{                                  … multi-line expression
runST (do
    ref <- newSTRef "hello"
    x <- readSTRef ref
    writeSTRef ref (x ++ "world")
    readSTRef ref )
:}
"helloworld"                            … result
```

http://hackage.haskell.org/package/base-4.12.0.0/docs/Data-STRef.html

STRef methods example (2)

modifySTRef :: STRef s a -> (a -> a) -> ST s ()

Mutate the contents of an **STRef**

```
>>> :{                                   … multi-line expression
runST (do
   ref <- newSTRef ""
   modifySTRef ref (const "world")
   modifySTRef ref (++ "!")
   modifySTRef ref ("Hello, " ++)
   readSTRef ref )
:}

"Hello, world!"                          … result
```

http://hackage.haskell.org/package/base-4.12.0.0/docs/Data-STRef.html

stToIO related methods (1)

```
stToIO        :: ST RealWorld a -> IO a
stToIO (ST m) = IO m

ioToST        :: IO a -> ST RealWorld a
ioToST (IO m) = (ST m)

unsafeSTToIO :: ST s a -> IO a
unsafeSTToIO (ST m) = IO (unsafeCoerce# m)

unsafeIOToST :: IO a -> ST s a
unsafeIOToST (IO m) = ST $ \ s -> (unsafeCoerce# m) s
```

(ST m)

(IO m)

(pattern matching)

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

stToIO related methods (1)

```
stToIO        :: ST RealWorld a -> IO a
stToIO (ST m) = IO m

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

ioToST        :: IO a -> ST RealWorld a
ioToST (IO m) = (ST m)

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

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

stToIO related methods (2) – safe and unsafe

The **safe** versions must start in the **IO monad**

- cannot obtain an **ST RealWorld** from **runST**)
- switch between the **IO** context and a **ST RealWorld** context.
- **safe** because **ST RealWorld** is basically the same thing as **IO**

The **unsafe** versions can start anywhere

- **runST** can be called anywhere
- switch between an arbitrary **ST monad** and the **IO monad**
- Using **runST** from a **pure** context and then doing a **unsafeIOToST**
- within the state monad is basically
  equivalent to using **unsafePerformIO**.

stToIO :: ST RealWorld a -> IO a

ioToST :: IO a -> ST RealWorld a

unsafeIOToST :: IO a -> ST s a

unsafeSTToIO :: ST s a -> IO a

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

stToIO related methods (3) – ioToST

At least, it will be ill-typed if you use **runST**.

notice that **ioToST** gives you an **ST RealWorld a**.

roughly speaking, **IO x** ≈ **ST RealWorld x**.

but **runST** won't accept **IO x** ≈ **ST RealWorld x** as input.

so you can't use **runST** to run I/O.

The **ioToST** gives you a type that cannot be used with **runST**.

But **unsafeIOToST** gives you a type that works just fine with **runST**.

At that point, you have basically implemented **unsafePerformIO**:

unsafePerformIO = runST . ioToST

stToIO :: ST RealWorld a -> IO a

ioToST :: IO a -> ST RealWorld a

unsafeIOToST :: IO a -> ST s a

unsafeSTToIO :: ST s a -> IO a

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

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

stToIO related methods (4) – escaping example

The **unsafeSTToIO** allows you
to get **mutable stuff** out of one **ST** **block**,
and potentially into another:

```
foobar = do
  v <- unsafeSTToIO (newSTRef 42)
  let w = runST (readSTRef v)
  let x = runST (writeSTRef v 99)
  print w
```

Because the thing is, we've got three **ST actions** here,
which can happen in absolutely any order.
Will the **readSTRef** happen before or after the **writeSTRef**?

```
unsafeIOToST :: IO a -> ST s a
unsafeSTToIO :: ST s a -> IO a

newSTRef :: a -> ST s (STRef s a)
readSTRef :: STRef s a -> ST s a
writeSTRef :: STRef s a -> a -> ST s ()

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

v :: STRef s a
```

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

stToIO related methods (5) – triggering IO operations

[Actually, in this example, the write never happens,

because we don't "do" anything with **x**.

But if I pass **x** to some distant and unrelated part of the code,

and if that code happens to inspect (evaluate) it,

suddenly our I/O operation does something different.

Pure code shouldn't be able to affect mutable stuff like that!]

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

ST Monad – source codes

```
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) }})
```

```
instance Functor (ST s) where
    fmap f (ST m) = ST $ \ s ->
      case (m s) of { (# new_s, r #) ->
      (# new_s, f r #) }

instance Applicative (ST s) where
    {-# INLINE pure #-}
    {-# INLINE (*>)   #-}
    pure x = ST (\ s -> (# s, x #))
    m *> k = m >>= \ _ -> k
    (<*>) = ap
    liftA2 = liftM2
```

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

runST (1)

```
{-# INLINE runST #-}
-- | Return the value computed by a state transformer computation.
-- The @forall@ ensures that the internal state used by the 'ST'
-- computation is inaccessible to the rest of the program.

runST :: (forall s. ST s a) -> a
runST (ST st_rep) = case runRW# st_rep of (# _, a #) -> a
-- See Note [Definition of runRW#] in GHC.Magic



data STRef s a = STRef (MutVar# s a)
newtype ST s a = ST (State# s -> (# State# s, a #))
```

http://hackage.haskell.org/package/base-4.12.0.0/docs/src/GHC.ST.html#ST

runST (2)

```
runST :: (forall s. ST s a) -> a
runST (ST st_rep) = case runRW# st_rep of (# _, a #) -> a

st_rep ::  State# s -> (# State# s, a #)
runRW# st_rep :: (# State# RealWorld, a #)
(# _, a #) :: (# State# RealWorld, a #)

runRW# ::  (State# RealWorld -> o) -> o    * memorization purpose

data STRef s a = STRef (MutVar# s a)
newtype ST s a = ST (State# s -> (# State# s, a #))
```

(ST st_rep)   (pattern matching)

http://hackage.haskell.org/package/base-4.12.0.0/docs/src/GHC.ST.html#ST

runRW# (1)

```
-- | Apply a function to a 'State# RealWorld' token. When manually applying
-- a function to `realWorld#`, it is necessary to use `NOINLINE` to prevent
-- semantically undesirable floating. `runRW#` is inlined, but only very late
-- in compilation after all floating is complete.

-- 'runRW#' is representation polymorphic: the result may have a lifted or
-- unlifted type.

runRW# :: forall (r :: RuntimeRep) (o :: TYPE r).
         (State# RealWorld -> o) -> o
```

http://hackage.haskell.org/package/ghc-prim-0.5.3/docs/src/GHC.Magic.html#runRW%23

runRW# (2)

```
-- See Note [runRW magic] in MkId
#if !defined(__HADDOCK_VERSION__)
runRW# m = m realWorld#
#else
runRW# = runRW#   -- The realWorld# is too much for haddock
#endif
{-# NOINLINE runRW# #-}
-- This is inlined manually in CorePrep
```

http://hackage.haskell.org/package/ghc-prim-0.5.3/docs/src/GHC.Magic.html#runRW%23

TYPE, RuntimeRep (1)

data TYPE (a :: RuntimeRep) :: RuntimeRep -> Type

data RuntimeRep

GHC maintains a property that the **kind** of all inhabited types
(as distinct from type constructors or type-level data)
tells us the **runtime representation** of **values** of that **type**.

This **datatype** encodes the choice of **runtime value**.

Note that **TYPE** is parameterised by **RuntimeRep**;
this is precisely what we mean by the fact
that a type's **kind** encodes the **runtime representation**.

http://hackage.haskell.org/package/ghc-prim-0.5.3/docs/src/GHC.Magic.html#runRW%23

TYPE, RuntimeRep (2)

data TYPE (a :: RuntimeRep) :: RuntimeRep -> Type

data RuntimeRep

For boxed values (that is, values that are represented by a pointer),

a further distinction is made,

between lifted types (that contain ⊥),

and unlifted ones (that don't).

http://hackage.haskell.org/package/ghc-prim-0.5.3/docs/src/GHC.Magic.html#runRW%23

TYPE, RuntimeRep (3)

data TYPE (a :: RuntimeRep) :: RuntimeRep -> Type

data RuntimeRep

VecRep VecCount VecElem	a SIMD vector type
TupleRep [RuntimeRep]	An unboxed tuple of the given reps
SumRep [RuntimeRep]	An unboxed sum of the given reps
LiftedRep	lifted; represented by a pointer
UnliftedRep	unlifted; represented by a pointer
IntRep	signed, word-sized value
WordRep	unsigned, word-sized value
Int64Rep	signed, 64-bit value (on 32-bit only)
Word64Rep	unsigned, 64-bit value (on 32-bit only)
AddrRep	A pointer, but not to a Haskell value
FloatRep	a 32-bit floating point number
DoubleRep	a 64-bit floating point numbe

http://hackage.haskell.org/package/ghc-prim-0.5.3/docs/src/GHC.Magic.html#runRW%23

References

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

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