Monad P3 : STRef Mutable Variable (3C)

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

ST Monad

Monad (ST s)

Methods

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

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

return :: a -> ST s a

fail :: String -> ST s a

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

STRef Mutable Variable

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

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**

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

IO, ST Monads and STRef Variable

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

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

data STRef s a  = STRef (MutVar# s a)
```

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

ST Monad – no initial state parameter

there is no parameter for the **initial state** as in **State** monad

ST uses a different notion of state to **State**;

State allows you to **get** and **put** the current state,

ST provides an **interface** to **references** of the type **STRef**

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

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

State Monad – providing the initial state

```
get :: State s s

get = state $ \s -> (s, s)

runState (get) s0

runState (get) 1

(1,1)
```

Initial state s0 can be supplied either by **runState**

or by the **initial monadic value**

```
put :: s -> State s a

put s ::  State s a

put newState = state $ \_ -> ((), newState)

runState (put ns) s0

runState (put 5) 1

((),5)
```

Initial state s0 can be supplied either by **runState**

or by the **initial monadic value**

https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

Interface for a reference

- use the following **interfaces** (methods) to a **reference** (**STRef**)
- to create **references** of the type **STRef**,
- to provide an **initial value** and to **manipulate** them

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

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

Mapping from references to values

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

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

the **internal environment** of a **ST computation** **runST**
is not one **specific value**,
but a **mapping** from **references** to **values**. **a** … **STRef s a**

ST s **a** …….. values

ST s **(STRef s a)** …… references

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

Start with empty mapping

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

no need to provide an **initial state** to **runST**, …….. **s**

as the **initial state** is

just the **empty mapping** containing no **references**.

Start with an **empty** mapping
– **no reference**

```
runST (do
   ref <- newSTRef "hello"
   x <- readSTRef ref
   writeSTRef ref (x ++ "world")
   readSTRef ref )
```

no ST computation should be allowed to assume that

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

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

Reference in a **ST** computation

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**

Example: Bad ST code

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

Two ST computations

Valid ST code

```
runST (do
    ref <- newSTRef "hello"
    x <- readSTRef ref
    writeSTRef ref (x ++ "world")
    readSTRef ref )
```

One ST computations

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

ST Monad Usage Example

```
import Data.STRef
import Control.Monad
import Data.Vector.Unboxed.Mutable as M
import Data.Vector.Unboxed as V

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

```
makeArray = runST $ do
   n <- newSTRef [1,2,3]
   readSTRef n

makeArray' = newSTRef 10 >>= readSTRef

makeArray'' = do
        a <- newSTRef 10
        b <- newSTRef 11
        return (a,b)



makeVec = runST $ do
        v <- M.replicate 3 (1.2::Double)
        write v 1 3.1
        V.freeze v
```

http://www.philipzucker.com/simple-st-monad-examples/

Example **sumST**

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

sumST :: Num a => [a] -> a
sumST xs = runST $ do                -- runST takes stateful ST code and makes it pure.
   summed <- newSTRef 0              -- Create an STRef (a mutable variable)

   forM_ xs $ \x -> do               -- For each element of the argument list xs ..
      modifySTRef summed (+x)        -- add it to what we have in n.

   readSTRef summed                  -- read the value of n, which will be returned by the runST above.
```

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

No leak information about **n** and **s**

sumST is no less **pure** than the familiar **sum**.

The fact that it destructively updates its accumulator **n**
is a mere implementation detail,

there is no way information about **n** could leak
except through the final result.

the **s** type variable in **ST s a** does not correspond to anything
in particular within the computation −
it is just an **artificial marker**.

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

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

n :: STRef s a
```

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

forM_ from the left

even though **forM_** folds the list from the right

the **sums** are done from the left,

as the mutations are performed as **applicative** effects

sequenced **from left to right**.

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

```
modifySTRef :: STRef s a ->
                        (a -> a) -> ST s ()
n :: STRef s a
(+x) :: a -> a
modifySTRef n (+x) :: ST s ()
```

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

Return value

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

return the **value** computed
by a **state transformer computation**.

The **forall** ensures that the **internal state** … **s**
used by the **ST computation** … **runST $ do …**
is inaccessible to the rest of the program.

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

n :: STRef s a

readSTRef :: STRef s a -> ST s a

readSTRef n :: ST s a
do ... readSTRef n :: ST s a
runST $ do … readSTRef n :: a

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

STRef – not strict modifying

Be warned that **modifySTRef** does not **apply** the function **strictly**.

if the program calls **modifySTRef** many times,

but seldomly uses the **value**,

thunks will pile up in memory resulting in a **space leak**.

do not use an **STRef** as a **counter**.

Memory leak example (likely produce a stack overflow)

```
print $ runST $ do
  ref <- newSTRef 0
  replicateM_ 1000000 $ modifySTRef ref (+1)
  readSTRef ref
```

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

Operations on **MutVar#'s**

data MutVar# s a

A MutVar# behaves like a single-element mutable array.

newMutVar# :: a -> State# s -> (#State# s, MutVar# s a#)

Create MutVar# with specified initial value in specified state thread.

readMutVar# :: MutVar# s a -> State# s -> (#State# s, a#)

Read contents of MutVar#. Result is not yet evaluated.

writeMutVar# :: MutVar# s a -> a -> State# s -> State# s

Write contents of MutVar#.

sameMutVar# :: MutVar# s a -> MutVar# s a -> Bool

https://downloads.haskell.org/~ghc/7.2.2/docs/html/libraries/ghc-prim-0.2.0.0/GHC-Prim.html#t:State-35-

Operations on **MutVar#**'s – source

```
data MutVar# s a

newMutVar# :: a -> State# s -> (# State# s,MutVar# s a #)
newMutVar# = let x = x in x

readMutVar# :: MutVar# s a -> State# s -> (# State# s,a #)
readMutVar# = let x = x in x

writeMutVar# :: MutVar# s a -> a -> State# s -> State# s
writeMutVar# = let x = x in x

sameMutVar# :: MutVar# s a -> MutVar# s a -> Bool
sameMutVar# = let x = x in x
```

https://downloads.haskell.org/~ghc/7.2.2/docs/html/libraries/ghc-prim-0.2.0.0/src/GHC-Prim.html#MutVar%23

STRef Definition (1)

```
data STRef s a = STRef (MutVar# s a)
-- ^ a value of type @STRef s a@ is a mutable variable in state thread @s@,
-- containing a value of type @a@
--
-- >>> :{
-- runST (do
--     ref <- newSTRef "hello"
--     x <- readSTRef ref
--     writeSTRef ref (x ++ "world")
--     readSTRef ref )
-- :}
-- "helloworld"
```

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

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

STRef Definition (2)

```
- | A value of type `STRef s a` is a mutable variable in state thread `s`,
-- containing a value of type `a`
data STRef s a = STRef (MutVar# s a)


-- |Build a new 'STRef' in the current state thread
newSTRef :: a -> ST s (STRef s a)
newSTRef init = ST $ \s1# ->
    case newMutVar# init s1# of
        { (# s2#, var# #)  ->  (# s2#, STRef var# #) }
```

https://osa1.net/posts/2016-07-25-IORef-STRef-exposed.html

STRef Definition (3)

```
-- | Read the value of an 'STRef'
readSTRef :: STRef s a -> ST s a
readSTRef (STRef var#) = ST $ \s1# -> readMutVar# var# s1#

-- | Write a new value into an 'STRef'
writeSTRef :: STRef s a -> a -> ST s ()
writeSTRef (STRef var#) val = ST $ \s1# ->
    case writeMutVar# var# val s1#  of
        { s2#    ->    (# s2#, () #) }
```

Note that there's no atomicModifySTRef, because that only makes sense in IO context. So atomicModifyIORef directly calls the primop.

https://osa1.net/posts/2016-07-25-IORef-STRef-exposed.html

newSTRef method (1)

```
-- |Build a new 'STRef' in the current state thread
newSTRef :: a -> ST s (STRef s a)
newSTRef init = ST $ \s1# ->
   case newMutVar# init s1# of
      { (# s2#, var# #)  ->  (# s2#, STRef var# #) }

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

s1# is transformed into s2#
which is embedded into var#

```
init :: a
s1# :: State# s
s2# :: State# s
var# :: MutVar# s a
```

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

newSTRef method (2)

```
newSTRef :: a -> ST s (STRef s a)
newSTRef init = ST $ \s1# ->
   case newMutVar# init s1# of
      { (# s2#, var# #)  ->  (# s2#, STRef var# #) }
```

```
newMutVar# :: a -> State# s -> (# State# s, MutVar# s a #)
newMutVar# init s1# ::  (# State# s, MutVar# s a #)
        init :: a
        s1# ::  State# s
```

```
init :: a
s1# ::  State# s
s2# ::  State# s
var# :: MutVar# s a
```

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

newSTRef method (3)

```
newSTRef :: a -> ST s (STRef s a)
newSTRef init = ST $ \s1# ->
   case newMutVar# init s1# of
      { (# s2#, var# #)  ->  (# s2#, STRef var# #) }
```

(pattern matching)

```
newMutVar# :: a -> State# s -> (# State# s, MutVar# s a #)
newMutVar# init s1# ::  (# State# s, MutVar# s a #)
(# s2#, var# #) ::  (# State# s, MutVar# s a #)
      s2# ::  State# s
      var# :: MutVar# s a
```

```
init :: a
s1# ::  State# s
s2# ::  State# s
var# :: MutVar# s a
```

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

newSTRef method (4)

```
newSTRef :: a -> ST s (STRef s a)
newSTRef init = ST $ \s1# ->
    case newMutVar# init s1# of
        { (# s2#, var# #)  ->  (# s2#, STRef var# #) }
```

```
newSTRef init = ST $ \s1# -> (# s2#, STRef var# #)
```

```
data STRef s a = STRef (MutVar# s a)
STRef var# :: STRef s a
```

```
newtype ST s a = ST (State# s -> (# State# s, a #))
ST $ \s1# -> (# s2#, STRef var# #) ::  ST s (STRef s a)
```

memorization purpose

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

newSTRef method (5)

```
newSTRef :: a -> ST s (STRef s a)
newSTRef init = ST $ \s1# ->
    case newMutVar# init s1# of
        { (# s2#, var# #)  ->  (# s2#, STRef var# #) }
```

```
newSTRef init = ST $ \s1# -> (# s2#, STRef var# #)
newSTRef init :: ST s (STRef s a)
```

s1# is transformed into **s2#**
which is embedded into **var#**

memorization purpose

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

readSTRef method (1)

```
-- | Read the value of an 'STRef'
readSTRef :: STRef s a -> ST s a
readSTRef (STRef var#) = ST $ \s1# -> readMutVar# var# s1#
```

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

s1# is transformed into **s2#**
extract the embedded value in **var#**

s1# :: State# s

var# :: MutVar# s a

(STRef var#)

(pattern matching)

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

readSTRef method (2)

```
readSTRef :: STRef s a -> ST s a
readSTRef (STRef var#) = ST $ \s1# -> readMutVar# var# s1#

data STRef s a = STRef (MutVar# s a)
STRef (MutVar# s a) :: STRef s a
STRef var# :: STRef s a
      var# :: MutVar# s a
```

(pattern matching)

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

readSTRef method (3)

```
readSTRef :: STRef s a -> ST s a
readSTRef (STRef var#) = ST $ \s1# -> readMutVar# var# s1#

newtype ST s a = ST (State# s -> (# State# s, a #))
ST (State# s -> (# State# s, a #)) :: ST s a
ST $ \s1# -> readMutVar# var# s1# :: ST s a
        s1# :: State# s
        readMutVar# var# s1# :: (# State# s, a #)

readMutVar# :: MutVar# s a -> State# s -> (# State# s, a #)
        var# :: MutVar# s a
```

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

readSTRef method (4)

```
readSTRef :: STRef s a -> ST s a
readSTRef (STRef var#) = ST $ \s1# -> readMutVar# var# s1#

readMutVar# :: MutVar# s a -> State# s -> (# State# s, a #)
        var# :: MutVar# s a

readMutVar# var# s1# = (# s2# val #)
        val :: a
```

memorization purpose

val is the embedded value in **var#**

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

readSTRef method (5)

```
readSTRef :: STRef s a -> ST s a
readSTRef (STRef var#) = ST $ \s1# -> readMutVar# var# s1#
```

```
readSTRef (STRef var#) = ST $ \s1# -> (# s2# , val #)
readSTRef (STRef var#) :: ST s a
```

memorization purpose

s1# is transformed into **s2#**
extract the embedded value in **var#**

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

writeSTRef method (1)

```
-- |Write a new value into an 'STRef'
writeSTRef :: STRef s a -> a -> ST s ()
writeSTRef (STRef var#) val = ST $ \s1# ->
   case writeMutVar# var# val s1# of
      { s2#    ->    (# s2#, () #) }


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

s1# :: State# s

s2# :: State# s

var# :: MutVar# s a

val :: a

(STRef var#)

(pattern matching)

s1# is transformed into s2#
change the embedded value in var#

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

writeSTRef method (2)

```
writeSTRef :: STRef s a -> a -> ST s ()
writeSTRef (STRef var#) val = ST $ \s1# ->
   case writeMutVar# var# val s1#  of
      { s2#    ->   (# s2#, () #) }
```

(pattern matching)

```
writeMutVar# :: MutVar# s a -> a -> State# s -> State# s
writeMutVar# var# val s1# :: State# s
      var# :: MutVar# s a
      val :: a
      s1# :: State# s
```

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

writeSTRef method (3)

```
writeSTRef :: STRef s a -> a -> ST s ()
writeSTRef (STRef var#) val = ST $ \s1# ->
   case writeMutVar# var# val s1#  of
      { s2#    ->   (# s2#, () #) }


writeMutVar# :: MutVar# s a -> a -> State# s -> State# s
writeMutVar# var# val s1# :: State# s
      s2# :: State# s
```

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

writeSTRef method (4)

```
writeSTRef :: STRef s a -> a -> ST s ()
writeSTRef (STRef var#) val = ST $ \s1# ->
   case writeMutVar# var# val s1#  of
      { s2#    ->   (# s2#, () #) }
```

```
writeSTRef (STRef var#) val = ST $ \s1# ->  (# s2#, () #)
```

```
newtype ST s a = ST (State# s -> (# State# s, a #))
ST $ \s1# -> (# s2#, () #) ::  ST s ()
```

memorization purpose

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

writeSTRef method (5)

```
writeSTRef :: STRef s a -> a -> ST s ()
writeSTRef (STRef var#) val = ST $ \s1# ->
    case writeMutVar# var# val s1#  of
        { s2#    ->    (# s2#, () #) }
```

```
writeSTRef (STRef var#) val = ST $ \s1# ->  (# s2#, () #)
writeSTRef :: STRef s a -> a -> ST s ()
```

memorization purpose

s1# is transformed into **s2#**
change the embedded value in **var#**

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

Summary (1)

```
newSTRef init = ST $ \s1# -> (# s2#, STRef var# #)
newSTRef init :: ST s (STRef s a)
```

memorization purpose

s1# is transformed into s2#
which is embedded into var#

```
readSTRef (STRef var#) = ST $ \s1# -> (# s2# , val #)
readSTRef (STRef var#) :: ST s a
```

memorization purpose

s1# is transformed into s2#
extract the embedded value in var#

```
writeSTRef (STRef var#) val = ST $ \s1# ->  (# s2#, () #)
writeSTRef :: STRef s a -> a -> ST s ()
```

memorization purpose

s1# is transformed into s2#
change the embedded value in var#

```
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 #))
```

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

Summary (2)

```
newSTRef init = ST $ \s1# -> (# s2#, STRef var# #)
newMutVar# :: a -> State# s -> (# State# s, MutVar# s a #)
```

```
readSTRef (STRef var#) = ST $ \s1# -> (# s2# , val #)
readMutVar# :: MutVar# s a -> State# s -> (# State# s, a #)
```

```
writeSTRef (STRef var#) val = ST $ \s1# ->  (# s2#, () #)
writeMutVar# :: MutVar# s a -> a -> State# s -> State# s
```

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

memorization purpose

s1# is transformed into s2#
which is embedded into var#

memorization purpose

s1# is transformed into s2#
extract the embedded value in var#

memorization purpose

s1# is transformed into s2#
change the embedded value in var#

```
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.STRef.html#STRef

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**

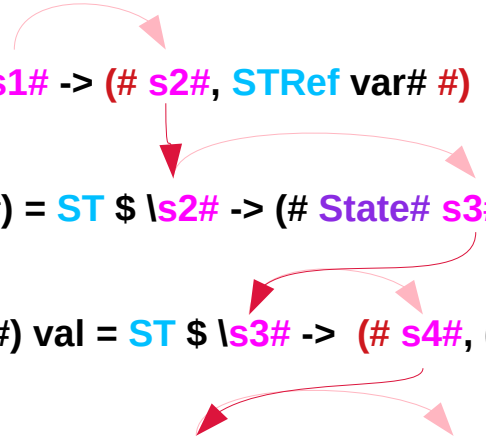
s1#, s2#, s3#, …

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

testST example – imperative style

```
runST (do
   ref <- newSTRef 0
   x <- readSTRef ref
   writeSTRef ref (x + 3)
   readSTRef ref )
```

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



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

Instance **Eq**

```
-- | Pointer equality.
--
-- @since 2.01
instance Eq (STRef s a) where
    STRef v1# == STRef v2# = isTrue# (sameMutVar# v1# v2#)
```

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

MutVar

data MutVar s a

A MutVar behaves like a single-element mutable array
associated with a primitive state token.

Constructors

MutVar (MutVar# s a)

http://hackage.haskell.org/package/primitive-0.7.0.0/docs/Data-Primitive-MutVar.html

MutVar Methods

newMutVar :: PrimMonad m => a -> m (MutVar (PrimState m) a)

Create a new MutVar with the specified initial value

readMutVar :: PrimMonad m => MutVar (PrimState m) a -> m a

Read the value of a MutVar

writeMutVar :: PrimMonad m => MutVar (PrimState m) a -> a -> m ()

Write a new value into a MutVar

http://hackage.haskell.org/package/primitive-0.7.0.0/docs/Data-Primitive-MutVar.html

PrimState m

class Monad m => PrimMonad m where

Class of monads which can perform primitive state-transformer actions

Associated Types

type PrimState m

state token type

Methods

primitive :: (State# (PrimState m) -> (#State# (PrimState m), a#)) -> m a

Execute a primitive operation

http://hackage.haskell.org/package/primitive-0.7.0.0/docs/Control-Monad-Primitive.html#t:PrimMonad

STRef Definition

class Monad m => PrimMonad m where

Class of monads which can perform primitive state-transformer actions

Associated Types

type PrimState m

State token type

Methods

primitive :: (State# (PrimState m) -> (#State# (PrimState m), a#)) -> m a

Execute a primitive operation

http://hackage.haskell.org/package/primitive-0.7.0.0/docs/Control-Monad-Primitive.html#t:PrimMonad

References

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

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