Monad P3 : Mutable Variables (2A)

Young Won Lim
2/18/20

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

This document was produced by using LibreOffice/OpenOffice.

Based on

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

Mutable State

Functional purity is a defining characteristic of Haskell,
mutable state can be avoided by the followings

- the **State monad** allows us to *keep track of* **state** in a convenient and **functionally pure way**
- efficient **immutable data structures** like the ones provided by the **containers** and **unordered-containers** packages

However, under some circumstances
using **mutable state** is just the most sensible option.

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

Program without mutable state – tail recursion

In C, you use **mutable variables** to create **loops** (like a **for** loop).

In Haskell, you can use **recursion**

to **re-bind** argument symbols in a **new scope**

(call the function with different arguments to get different results).

Problem: recursive factorial implementation

each function call creates stack frames

thus eventually memory is wasted

Solution: Haskell supports **optimized tail recursion.**

Use an **accumulator argument**

https://www.scs.stanford.edu/14sp-cs240h/notes/00-getting-started/basics.html

Program without mutable state – guards, where clauses

Guards let you **shorten** function declarations

by **declaring conditions** in which a function occurs:

pipe ("|") symbol introduces a **guard**.

Guards are evaluated top to bottom

the first True guard wins.

otherwise in the Haskell system Prelude evaluates to true

Bindings can end with **where** clauses

where clauses can scope over multiple guards

convenient for **binding variable**s to use in guards

https://www.scs.stanford.edu/14sp-cs240h/notes/00-getting-started/basics.html

guards, where clause examples

```
absolute x
  | x<0 = -x
  | otherwise = x



holeScore :: Int -> Int -> String
holeScore strokes par
  | score < 0       = show (abs score) ++ " under par"
  | score == 0      = "level par"
  | otherwise       = show(score) ++ " over par"
 where score = strokes-par
```

https://www.futurelearn.com/courses/functional-programming-haskell/0/steps/27226

Purely functional

Haskell is a **purely functional** language:

there are **no side-effects** and

all variables are **immutable**.

All variables are indeed **immutable**,

but there are ways to construct **mutable references**

where we can change what the **reference** points to.

https://blog.jakuba.net/2014/07/20/Mutable-State-in-Haskell/

Side effects and Mutable state

Without **side effects** we wouldn't be able to do much,
which is why Haskell gives us the **IO monad**.

In a similar manner we have many ways
to achieve **mutable state** in Haskell

- **IORef** in the **IO monad** — **mutable reference**
- **STRef** in the **ST monad** — **mutable reference**
- **MVar**
- **TVar** in Software Transactional Memory (STM)

https://blog.jakuba.net/2014/07/20/Mutable-State-in-Haskell/

Mutable Variables

the functional programming

immutable variables

mutable variables are needed sometimes

1) simulate **mutable variables**

2) use real **mutable variables**

In either case you need a **monad**

in order to deal with **mutability**,

while staying **purely functional**.

http://wiki.haskell.org/Mutable_variable

State Monad and IORef and STRef Mutable Variables

simulating mutable variables

State monad

in **Control.Monad.Trans.State**

from the **transformers package**

using real mutable variables

IORef or **STRef**

Data.IORef or **Data.STRef** or

Control.Concurrent.STM.TVar

from the **STM** package.

http://wiki.haskell.org/Mutable_variable

Mutable Variables

Mutability is not actually expressed through **monads**.

Monads are a much more general way of
composing computations. **bind operator >>=**

It happens to be useful in
composing computations for **mutation**.

https://www.quora.com/How-does-Haskell-express-mutability-through-monads

Versioning

mutation is not really necessary in most computations

Versioning can almost always replace
(Single-threaded) **mutation** of data structures

create a new version of data
by making a **clone** of it,
which contains the mutated part.

https://www.quora.com/How-does-Haskell-express-mutability-through-monads

Versioning and Pointers

Instead of mutating the head of a list, for instance,

you make a new list with the new head and the same tail.

But since all data structures are **immutable** in Haskell,

the **creation** of the new list does not involve any **copying**

the compiler will just use a **pointer**

to the existing (immutable) tail.

https://www.quora.com/How-does-Haskell-express-mutability-through-monads

Copying arguments

Haskell's libraries contain all kinds of **data structures**
that can be easily modified without **mutation**.
They are called **persistent data structures**.

In Haskell, a **function** that would traditionally **mutate** its **argument**,
will explicitly return a **new modified copy** of **argument**

The following function takes an integer and returns an integer.
By the type it cannot do any side-effects whatsoever,
it cannot mutate any of its arguments.

```
square :: Int -> Int
square x = x * x
```

https://www.quora.com/How-does-Haskell-express-mutability-through-monads

Persistent data structures

In computing, a **persistent data structure** is a data structure that always preserves the previous version of itself when it is modified.

Such data structures are effectively immutable, as their operations do not (visibly) update the structure in-place, but instead always yield a new updated structure.

https://www.quora.com/How-does-Haskell-express-mutability-through-monads

Persistent data structures in Haskell

all data structures in the language are **persistent**,
as it is impossible to not preserve the previous state
of a data structure with **functional semantics**.

This is because any change to a data structure
that would render previous versions of a data structure invalid
would violate **referential transparency**.

In its standard library Haskell has
efficient persistent implementations for

- **Linked lists**,
- **Maps** (implemented as size balanced trees), and
- **Sets**

https://www.quora.com/How-does-Haskell-express-mutability-through-monads

Referential transparency

An **expression** is called **referentially transparent**
if it can be replaced with its corresponding **value**
without changing the program's behavior.

This requires that the **expression** be **pure**,
that is to say the **expression** value must be
the same value for the same inputs
and its evaluation must have no side effects.

https://www.quora.com/How-does-Haskell-express-mutability-through-monads

Argument to functions

In **imperative languages**,

mutation is often <u>hidden</u> from the type system.

Instead of passing and returning a **modifiable argument**,

procedures secretly access **external state** (**global variables**) or

mutate the **arguments** that are passed to it **by reference**.

call by reference

In **Haskell**, this is impossible: All functions are **pure**.

So if something has to be modified,

it must <u>appear</u> in the **function's signature**,

both as **input** (argument) and **output** (part of the return type).

https://www.quora.com/How-does-Haskell-express-mutability-through-monads

Composing functions

This leads to the necessity of **composing functions**
that return those enriched values whose enrichment
is necessary to transmit the **new state**.

A naive approach to this would entail a lot of **boilerplate code**.

The **monad** instead provides a streamlined way of
organizing these repetitive tasks.

It allows you to ***define composition*** in one place
and then use it to ***create longer sequences***
of **stateful computations**.

s -> (s, a)

(s, a) : enriched values

https://www.quora.com/How-does-Haskell-express-mutability-through-monads

Do notation

Together with the syntactic sugar of the **do** notation,
function composition makes for a very concise programming style
that enables to imitate **mutability**.

mutations are **encapsulated** in the **state** data structure,
and **composition** automatically combines
state modifications performed by individual **functions**.

The **do**-notation even hides the **state** from view.
But since the code is still **pure**, you may, for instance,
safely use it in **parallel programming**, without any fear of data races.

state threading

```
main = do
  box <- newIORef (42 :: Int)
  num <- readIORef box
  print num
  writeIORef box 0
  num <- readIORef box
  print num
```

https://www.quora.com/How-does-Haskell-express-mutability-through-monads

Boiler plate code

boilerplate code are
sections of code that have to be included
in many places with little or no alteration.

When using languages that are considered verbose,
the programmer must write a lot of code
to accomplish only minor functionality.

https://en.wikipedia.org/wiki/Boilerplate_code

State Threading

Through **state-threading**. In Haskell,

there are only **expressions**, no **statements**.

Assume that **expressions** depend solely on their **arguments**.

This makes **gettime()** a little challenging.

What should it return?

And what should we pass it?

If we pass it nothing,

we can call clearly only **gettime()** once.

referential transparency

https://www.quora.com/How-does-Haskell-express-mutability-through-monads

State tokens

To resolve this difficulty, Haskell models **side effects**
by passing a **state token**;

every **side-effecting function**
accepts and receives a **state token**.

This is rather like
multi-view concurrency control in a database,
every query can be seen as
taking place with a certain **transaction ID** and
also generating a new **transaction ID**.

a **transaction ID** is like a **state token**

s -> (s, a)

s : state token

https://www.quora.com/How-does-Haskell-express-mutability-through-monads

Side-effecting functions with a state token

getTime :: StateToken -> (StateToken, StructTime).

For **generalized IO**, the state token
is taken to be the state of the world and
is generated in such a way that each is **unique**;

this accounts for the type signature of the IO **monad**:

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

https://www.quora.com/How-does-Haskell-express-mutability-through-monads

IO monad’ state function

The **IO monad** is a special form of the **State monad**

more limited scopes for **state**

– a particular memory pool, a particular map

What **IO** captures is

a **function** from **State# RealWorld**

to a **tuple** of a **new state** and a **result**.

enriched value

(**State# RealWorld:** a low-level, **unboxed** type)

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

https://www.quora.com/How-does-Haskell-express-mutability-through-monads

Side-effecting function calls

Because each **state token** is **unique**,

and every **side-effecting call** requires one, it falls out that:

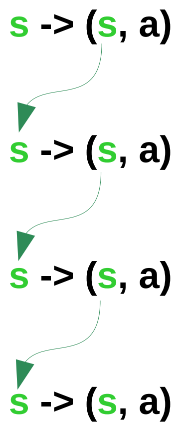
Side-effecting calls can not be eliminated or interchanged:

to the compiler, every such call is **unique**

so there is no unfortunate optimization of side-effects

Each call depends on the preceding one;

so there can be no reordering of these calls.



https://www.quora.com/How-does-Haskell-express-mutability-through-monads

Dependency and Uniqueness

Haskell is an **expression** oriented language;

and **expressions** depend on their **arguments**;

in general the compiler may

eliminate multiple identical calls or

reorder calls relative to one another.

However, when one expression depends on the output of the other,

the latter must wait for the first one to complete;

when two expressions are different (with different argument)

we can not remove one or the other.

a way to model arbitrary **side-effects**.

s -> (s, a)

s -> (s, a)

s -> (s, a)

s -> (s, a)

Dependency
Uniqueness

https://www.quora.com/How-does-Haskell-express-mutability-through-monads

State threading through >>= bind operator

Ensuring that <u>one</u> **side-effecting function**

wait for the **token** from <u>the other</u>

IO monad <u>simplifies</u> the **state-threading**

– users <u>need not to pass</u> the **state token**

– actually it is <u>protected</u> from <u>tampering</u>

This is done through a monad's

composition, **>>=** (**bind**).

https://www.quora.com/How-does-Haskell-express-mutability-through-monads

Composition example

a **monadic computation**

getTime returns a **StructTime**

a **function** **printTime** prints a **StructTime**

then you may do so as follows:

```
main = getTime >>= printTime

    getTime :: IO StructTime
    printTime :: StructTime -> IO ()
```

The **bind operation >>=** of **IO** takes care of

chaining the implicit state tokens.

https://www.quora.com/How-does-Haskell-express-mutability-through-monads

Mutable Variables

The only place where **mutation** is really needed is in **concurrent programming**, but even there it can be dealt with using the **IO monad**. But that's a separate topic.

https://www.quora.com/How-does-Haskell-express-mutability-through-monads

Mutable Reference Example

For example, let's create a **mutable reference** and **modify** it:

```
import Data.IORef

main = do
  box <- newIORef (42 :: Int)              :: IO (IORef Int)
  num <- readIORef box                     :: IO Int
  print num                                :: IO ()
  writeIORef box 0                         :: IO ()
  num <- readIORef box                     :: IO Int
  print num                                :: IO ()
```

https://www.quora.com/How-does-Haskell-express-mutability-through-monads

IO () type expression

some lines of this program after the **do** is
an **expression** of type **IO ()**

its evaluation will not do something,
like readin4g or writing the **IORef**,
but instead will return a **command** **thunks**
that the **IO Monad** can choose to execute.

The **Haskell runtime**
will take that **command** and
actually **execute** it.

https://www.quora.com/How-does-Haskell-express-mutability-through-monads

writeIORef and **readIORef**

two kinds of expressions:

writeIORef box 0 is of type **IO ()**

This expression does not change the **IORef**,

it returns the **command** to do that. **thunks** **monadic computation**

readIORef box is of type **IO Int**

it returns a **command** from whose execution **thunks** **monadic computation**

you can extract an **Int**.

https://www.quora.com/How-does-Haskell-express-mutability-through-monads

<- operator

```
num <- readIORef box                              :: IO Int
```

The **<-** operator returns an **IO Int**
whose **execution** does that extraction,
putting the **Int** in the named variable **num**

Actually, **IO ()** and **IO Int** aren't different,
but **()** is just the type of a singleton value that is called Unit.

So you could do **foo <- print 1** but it doesn't serve any purpose.

print 1 is of type **IO ()**, its evaluation returns
a **command** and has no side effect.

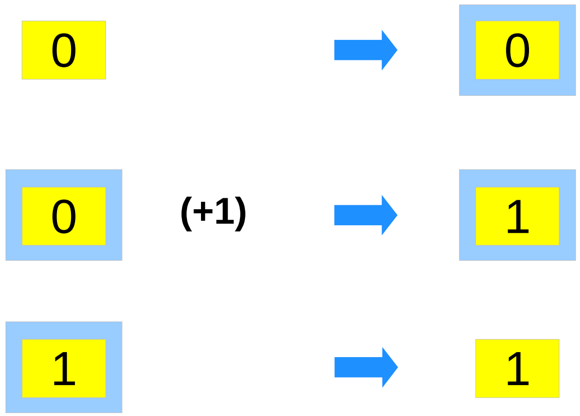
https://www.quora.com/How-does-Haskell-express-mutability-through-monads

IORef Usage

ref :: IORef a
ref <- newIORef 0

modifyIORef ref (+1)

readIORef ref




```
data IORef a
newIORef      :: a -> IO (IORef a)
modifyIORef  :: IORef a -> (a -> a) -> IO ()
readIORef     :: IORef a -> IO a
```

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

IORef Example

```
newIORef :: a -> IO (IORef a)
newIORef 0 :: IO (IORef a)
ref <- newIORef 0
ref :: IORef a

(+1) :: (a -> a)
modifyIORef :: IORef a -> (a -> a) -> IO ()
modifyIORef ref (+1) :: IO ()

readIORef :: IORef a -> IO a
readIORef ref :: IO a


ref <- newIORef 0
replicateM_ 1000000 $ modifyIORef ref (+1)
readIORef ref >>= print
```

```
data IORef a

newIORef     :: a -> IO (IORef a)
readIORef    :: IORef a -> IO a
writeIORef   :: IORef a -> a -> IO ()
modifyIORef  :: IORef a -> (a -> a) -> IO ()
modifyIORef‘ :: IORef a -> (a -> a) -> IO ()
```

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

IO, **ST** Monads and **IORef**, **STRef** Variables

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

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

newtype IORef a = IORef (STRef RealWorld 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

testST example – imperative style

```
runST (do
    ref <- newSTRef 0                newSTRef init = ST $ \s1# -> (# s2#, STRef var# #)

    x <- readSTRef ref               readSTRef (STRef var#) = ST $ \s2# -> (# State# s3#, val #)

    writeSTRef ref (x + 3)           writeSTRef (STRef var#) val = ST $ \s3# ->  (# s4#, () #)

    readSTRef ref )                  readSTRef (STRef var#) = ST $ \s4# -> (# State# s3#, val #)
```

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

State# s

data State# s

type definition without a data constructor

The primitive type **State#** represents
the **state** of a **state transformer**.

State# is the **primitive**, **unlifted** type of states.
It has one type parameter **s**,

State# RealWorld, or **State# s**,
where **s** is a type variable.

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

State# s – purpose

The only purpose of the **type parameter** s is

to keep different **state threads** separate.

It is represented by nothing at all.

It is parameterised on the desired **type of state**, (**s**)

which serves to keep states distinct **threads** distinct from one another.

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

State# s – effect

But the only effect of this **parameterisation** is in the **type system**:
all **values** of type **State#** are represented in the same way.

Indeed, they are all represented by nothing at all!

The code generator “knows”
to generate **no code**,
and allocate **no registers** etc,
for **primitive states**.

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

RealWorld

data **RealWorld**

type definition without a data constructor

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#**.

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

RealWorld – no values, no operations

The type GHC.**RealWorld** is truly opaque:

No **data constructor**

there are no **values** defined of this type,

and no **operations** over it.

It is “**primitive**” in that sense -

but it is not **unlifted**!

Its only role in life is to be the type

which distinguishes the **IO state transformer**.

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

RealWorld

the **RealWorld** tokens have type **State# RealWorld**,

which is yet another primitive type … of size 0.

So let us retrace our steps and consider the same examples

that we considered for **Int#** and **Double#**,

but now look at the corresponding translation for **State# RealWorld**.

We first considered the construction of a constant:

```
constant_State# :: () -> State# RealWorld
constant_State# _ = realWorld#
```

This translates to

```
Sp = Sp + 4;
jump (I32[Sp + 0]) ();
```

https://www.well-typed.com/blog/2014/06/understanding-the-realworld/

realWorld# – a reference to the real world

realWorld# is a **value** of type **State# RealWorld**

which is a **token** that acts as a **reference** to **the real world**.

(it is of **size** 0 and does not occupy any space

on the **stack** or **heap**.)

State# RealWorld values represent

the entire external **runtime state** of the program.

The "real world", as it were.

The **main** value in your program

receives a **State# RealWorld** **value**

that is threaded through the **IO actions** that compose it.

https://stackoverflow.com/questions/32672814/where-is-the-realworld-defined

State# RealWorld

The **primitive** **State# RealWorld**

RealWorld corresponds to the **s** parameter of our **State** monad

Actually, it's two **primitives**, the **type constructor** **State#**,
and the magic type **RealWorld** which doesn't have a **# suffix**

This is because **ST** monad also uses
a **type constructor** and a **type parameter** framework

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

State# RealWorld – type

You can treat **State# RealWorld** as a **type**

that represents a very **magical value**:

the value of the entire real world.

only the **main** function can receive a **real world value**,

and it then gets threaded through **sequence** of **IO actions**

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

MutVar# s a

MutVar# is a **primitive type**

It represents a **mutable reference**,

and is used by **IORef** and **STRef**.

In general, anything that ends in # is

an implementation detail of GHC.

Most of these operations have wrappers (like ST)

which are easier to use.

https://stackoverflow.com/questions/30448007/what-does-mutvar-mean

MutVar# s a

data MutVar# (a :: Type) (b :: Type) :: Type -> Type -> TYPE UnliftedRep

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

MutVar# s a

http://hackage.haskell.org/package/base-4.12.0.0/docs/GHC-Exts.html#t:MutVar-35-

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**.

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/base-4.12.0.0/docs/GHC-Exts.html#t:MutVar-35-

GHC.Prim

GHC is built on a raft of **primitive data types** and **operations**;

- primitive in the sense that

they cannot be defined in Haskell itself

- optimised to the efficient **unboxed** version

the **primitive** data types or operations **unboxed**

exported by the library **GHC.Prim**

have names ending in #

extensive use of **unboxed types** and **unboxed tuples**

https://downloads.haskell.org/~ghc/7.0.1/docs/html/users_guide/primitives.html

IO, IORef, ST, STRef, State Definitions

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

newtype  IORef   a    = IORef (STRef RealWorld a)
data         STRef s a    = STRef (MutVar# s a)

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

https://stackoverflow.com/questions/18295211/signature-of-io-in-haskell-is-this-class-or-data

(State s) Monad

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

instance Monad (State s) where
(>>=) :: State s a -> (a -> State s b) -> State s b
p >>= k = q where
    p' = runState p              -- p' :: s -> (a, s)
    k' = runState . k            -- k' :: a -> s -> (b, s)
```

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

(ST s) Monad

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

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) }     } )

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

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

IO Monad

IO Monad

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

instance  Monad IO  where
    return    = returnIO
    (>>=)     = bindIO

returnIO :: a -> IO a
returnIO x = IO $ \s -> (# s, x #)

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

GHC.Types

System.IO

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

IORef Mutable Variable

newtype IORef a = IORef (STRef RealWorld a)

STRef in the **IO** monad.

IORefs do not have the same **safety guarantees** as **STRef**s about **locality**.

It's just a **newtype wrapper** around a specialized **STRef RealWorld**, and the only thing it adds over **STRef** are some **atomic operations**.

https://stackoverflow.com/questions/5545517/difference-between-state-st-ioref-and-mvar

IORef and concurrency

In **non-concurrent code**, there's no good reason

not to use **STRef s** values in an **ST s** monad,

since they're more flexible --

you can run them in pure code with **runST** or,

if needed, in the **IO** monad with **stToIO**.

In **concurrent code**, there are more powerful abstractions,

like **MVar** and **STM** that are much easier

to work with than **IORefs**.

https://stackoverflow.com/questions/52467957/ioref-in-haskell

IORef vs STRef

IORef and STRef each provide the same functionality,
but for different monads.

IORef for IO Monad
STRef for ST Monad

Use IORef if you need a managed **ref** in IO,
and STRef if you need one in ST **s**.

```
newtype  IORef    a    = IORef (STRef RealWorld a)
data     STRef s a     = STRef (MutVar# s a)
```

https://stackoverflow.com/questions/20439316/when-to-use-stref-or-ioref

IORef vs STRef – examples

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

exampleSTRef :: ST s Int
exampleSTRef = do
  counter <- newSTRef 0
  modifySTRef counter (+ 1)
  readSTRef counter
```

```
import Control.Monad.ST
import Data.IORef

exampleIORef :: IO Int
exampleIORef = do
  counter <- newIORef 0
  modifyIORef counter (+ 1)
  putStrLn "im in ur IO monad so i can do I/O"
  readIORef counter
```

https://stackoverflow.com/questions/20439316/when-to-use-stref-or-ioref

ST Monad – the restricted version of **IO** Monad

The **ST** monad is

- the restricted version of the **IO** monad.
- the less dangerous sibling of the **IO** monad, or **IO computations**, where you can only read and write to memory.

https://stackoverflow.com/questions/5545517/difference-between-state-st-ioref-and-mvar

ST Monad – safety measures about locality

- **STRef**s have **safety** **guarantees** about **locality**

  IORefs do not have

  The API is made **safe** in **side-effect-free** programs,

  as the **rank-2 type parameter** prevents **values**

  that depend on **mutable state** from escaping **local** **scope**.

  thus allows for **controlled mutability**

  in otherwise **pure** programs.

https://stackoverflow.com/questions/5545517/difference-between-state-st-ioref-and-mvar

ST Monad – mutable state

- **ST** allows arbitrary **mutable state**,
  implemented as **actual mutable memory** on the machine.

- The **mutable state** of **ST** is very efficient
  since it is **hardware accelerated**

- commonly used for **mutable arrays** and
  other data structures that are **mutated**
  are frozen

https://stackoverflow.com/questions/5545517/difference-between-state-st-ioref-and-mvar

ST Monad – primary API

Control.Monad.ST

runST -- start a new memory-effect computation.

STRefs: pointers to (local) mutable cells.

ST-based arrays (such as **vector**) are also common.

https://stackoverflow.com/questions/5545517/difference-between-state-st-ioref-and-mvar

ST Monad – MVars

MVars : IORefs with locks

Like **STRefs** or **IORefs**, but with a **lock** attached,
for **safe concurrent access** from **multiple threads**.

MVars are a more general mechanism
for **safely sharing mutable state**.

use **MVars** or **TVars** (**STM**-based mutable cells),
over **STRef** or **IORef**. (specially in **concurrent** applications)

https://stackoverflow.com/questions/5545517/difference-between-state-st-ioref-and-mvar

ST Monad – atomic swap operation

MVars : IORefs with <u>locks</u>

IORefs and **STRef**s can be **safe**
in a **multi-threaded** (**concurrent**) applications
if **atomicModifyIORef** is used

(a **compare-and-swap atomic** operation).

https://stackoverflow.com/questions/5545517/difference-between-state-st-ioref-and-mvar

ST Monad – imperative code enabled

functions written using the **ST** **monad**

appear completely **pure** to the rest of the program.

Mutable variables **STRef**s allows programmers

to produce **imperative code**

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

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

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

ST Monad advantage

The **ST monad** allows programmers
to write **imperative algorithms** in Haskell,

by using mutable **variables** (**STRef**'s)
and mutable **arrays** (**STArrays** and **STUArrays**).

- **code** can have internal **side effects**
  - destructively updating mutable **variables** and **arrays**,
  - containing these **effects** inside the monad.

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

Imperative coding style using **STRef** Monad

While in place modifications of the **n**

of the typ**e STRef s a** are occurring,

something that would usually be considered a **side effect**,

it is all done in a safe way which is deterministic.

Memory modification in place is possible

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

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

ST Monad – imperative code example

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

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

a **variable** is directly updated, ….. imperative style

rather than a **new value** is formed and ….. functional style

passed to the **next iteration** of the function.

Imperative style code example

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

using a **mutable variable**:

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

State Monad – APIs

The **State** **Monad** : a **model** of **mutable state**

The **State monad** is a purely functional environment
for programs with **state**, with a simple **API**:

get
set the result value to the **state** and
leave the **state** unchanged.

put
set the result value to () and
set the **state** value. … *mutable*

Documentation in the **mtl** package.

https://stackoverflow.com/questions/5545517/difference-between-state-st-ioref-and-mvar

State Monad – the parameterized state

The **State monad** is commonly used
when needing **state** in a **single thread of control**.
(not **concurrent**)

It does **not** actually use **mutable state** in its implementation.

Instead, the program is parameterized by the **state value**
(i.e. the state is an additional parameter to all computations).

The **state** only *appears to be* mutated in a **single thread** ….. **put**
(and cannot be shared between threads).

https://stackoverflow.com/questions/5545517/difference-between-state-st-ioref-and-mvar

State and **ST**

Conceptually, the difference is in the **API**.

State can be thought of as
an **ST** with a single, implicit reference cell.

Alternately, **ST** can be thought of as
a **State** which manipulates a store of values.

https://mail.haskell.org/pipermail/haskell/2007-May/019540.html

ST and State Definitions

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

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

https://stackoverflow.com/questions/18295211/signature-of-io-in-haskell-is-this-class-or-data

State in terms of ST (1)

```
newtype State s a = State
  { runState :: forall r. ReaderT (STRef r s) (ST r) a }

runState :: State s a -> s -> (a,s)
runState m s0 = runST (do
  r <- newSTRef s0
  a <- runReaderT (unState m) r
  s <- readSTRef r
  return (a,s))
```

https://mail.haskell.org/pipermail/haskell/2007-May/019540.html

State in terms of ST (2)

```
instance Monad (State s) where
    return a = State (return a)
    m >>= f  = State (unState m >>= unState . f)

instance MonadState s (State s) where
    get   = State (ask >>= lift . readSTRef)
    put x = State (ask >>= \s -> lift (writeSTRef s x))
```

https://mail.haskell.org/pipermail/haskell/2007-May/019540.html

ST and State Definitions

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


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

https://stackoverflow.com/questions/18295211/signature-of-io-in-haskell-is-this-class-or-data

ST in terms of State (1)

Assume we have a Store ADT with this interface:

```
data Store r
data STRef r a
withStore :: (forall r. Store r -> a) -> a
newRef    :: a -> Store r -> (STRef r a, Store r)
readRef   :: STRef r a -> Store r -> a
writeRef  :: STRef r a -> a -> Store r -> Store r
```

(The 'r' parameter is to make sure that references are only used with the Store that created them. The signature of withStore effectively gives every Store a unique value for r.)

https://mail.haskell.org/pipermail/haskell/2007-May/019540.html

ST in terms of State (2)

```
newtype ST r a = ST { unST :: State (Store r) a } deriving Monad

runST :: (forall r. ST r a) -> a
runST m = withStore (evalState (unST m))

newSTRef :: a -> ST r (STRef r a)
newSTRef a = ST $ do
    s <- get
    let (r,s') = newRef a s
    put s'
    return r
```

https://mail.haskell.org/pipermail/haskell/2007-May/019540.html

ST in terms of State (3)

```
readSTRef :: STRef r a -> ST r a
readSTRef r = ST $ gets (readRef r)

writeSTRef :: STRef r a -> a -> ST r ()
writeSTRef r a = ST $ modify (writeRef r a)
```

https://mail.haskell.org/pipermail/haskell/2007-May/019540.html

Subtleties

There are two subtleties.

The first is that you can't implement Store
without cheating at some level (e.g., unsafeCoerce).

The second is that
the real ST implementation uses in-place update, which is only safe
because the Store is implicit and used single-threadedly.

https://mail.haskell.org/pipermail/haskell/2007-May/019540.html

:{ :} multi-line GHCi command block

:{

:}

begin or end a multi-line GHCi command block.

GHCi commands can be split over multiple lines,

by wrapping them in :{ and :} (each on a single line of its own):

```
Prelude> :{
Prelude| g op n [] = n
Prelude| g op n (h:t) = h `op` g op n t
Prelude| :}
Prelude> g (*) 1 [1..3]
6
```

https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/ghci.html

Haddock comment (1)

```
module Fib where
-- | Compute Fibonacci numbers
--
-- Examples:
--
-- >>> fib 10          …    expression
-- 55                  …    result
--
-- >>> fib 5           …    expression
-- 5                   …    result
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) + fib (n – 2)
```

https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/ghci.html

Haddock comment (2)

A comment line starting with **>>>** denotes an **expression**.

All comment lines following an **expression** denote
the **result** of that **expression**.

Result is defined by what an REPL (e.g. ghci) prints
to **stdout** and **stderr** when evaluating that expression.)

https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/ghci.html

References

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

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