Lightning Talk: Mutable Things in Haskell

SGFP: Singapore Functional Programming Group Meeting

Yang Ye

Runchee Technology

17 Oct 2013

- `yy@runchee.com`

Outline

Immutable

No self-assignment

```
let x = 1
    x = x + 1
print x
```

Immutable

Name can be recycled but moved to another (memory) location.

```
let x = 1
    x = 2
print x
```

Immutable

- Immutable is about purity
- Purity is how a programmer designs the code to separate pure and effects code
- Purity is what keeps compiler faster, generate optimised code.

Immutable v.s. Mutables

- What's good in mutables?
  - Convenience: in-place modification
  - Speed
- What's bad in mutables?
  - no problem in ST
  - many problems in MT

Application pattern

- Pattern 1:

1. get input from outside
2. process
3. sent output to outside

- Pattern 2:

1. ...
2. process: interact with outside
3. ...

Solution

- The process is a recursive function to transform state

```
f :: State -> ... -> State
```

Solution

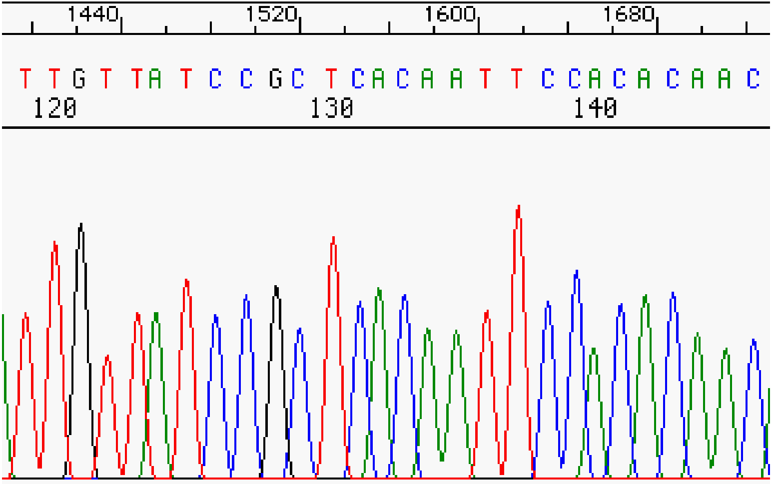
- Think and Plan:
  - How often it get changed?
  - What scope does it affected? limited, global
  - What's the requirement, need for speed?

Haskell's Solution

- Localized
  - State Monad
  - ST Monad
- Closed but operate from inside
  - StateT Monad Transformer
- Open
  - IORef
  - MVar
  - TVar (STM, not covered)

DNA Sequence Statistics

- DNA: A, T, G, C



DNA Sequence

```
CCCCTGATCGAATTCCTATTCTAGCCGCTCCTGCCGTTTTGC
GTCTGCGGCGTGTAGCGCGTATCACAGTTCGAGGTGGAATCA
CTGCACAAGGGATTAGTGGGAGATAGTATGGGGCGTGTCGGG
TTTAGTAGTGTCCGTGAAGCGAGCACACACTCTGAACGTATT
GCACATAGTACTTAGCAATCCTTCGCACCCCATGGTGTGGGA
TCGTGAACCGTAGCCGTGGGTAGATCTCTTCGATTGAGCGAA
....
```

A:XX%, T:YY%, G:ZZ%, C:WW%

Python code

```
seq = "ACCCCTGATCGA..."

d = { "A": 0, "C": 0, "G": 0, "T": 0 }

for nuc in seq:
    d[nuc]+=1

print d["A"], d["C"], d["G"], d["T"]
```

Haskell

```haskell
type SeqInfo = (Int, Int, Int, Int)
emptySeqInfo = (0,0,0,0)

countSeq :: Char -> SeqInfo -> SeqInfo
countSeq n w@(a,t,g,c) = case n of
    'A' -> (a+1,t,g,c)
    'T' -> (a,t+1,g,c)
    'G' -> (a,t,g+1,c)
    'C' -> (a,t,g,c+1)
    otherwise -> w

countNuc :: String -> SeqInfo
countNuc seq = foldl (flip countSeq) emptySeqInfo seq
```

Haskell - 2

```
main :: IO ()
main = do
    seq <- readFile "seq.dna"
    let (cA, cT, cG, cC) = countNuc seq
    putStrLn ("A:" ++ show cA ++ " T:" ++ show cT
          ++ " G:" ++ show cG ++ " C:" ++ show cC)
```

Classic example of pattern 1

State Monad

- State and its relative ST both produce ‘monolithic’ stateful computations which may be run as units.
- API: get, put, runState

```
State (String, SeqInfo)

countNuc2 :: String -> SeqInfo
countNuc2 seq = (cA, cT, cG, cC)
    where (s, (cA, cT, cG, cC)) =
              runState (calc seq) emptySeqInfo
              :: (String, (Int, Int, Int, Int))
          calc (x:xs) = do
            (a,t,g,c) <- get
            put $ countSeq x (a,t,g,c)
            calc xs
          calc [] = return ""
```

ST Monad

The mutable state is localized and do not require interaction with the environment.

- API
  - runST: start a new memory-effect computation.
  - STRefs: pointers to (local) mutable cells.
  - ST-based arrays

```
countNuc2ST :: String -> SeqInfo
countNuc2ST seq = runST $ do
    cseq <- newSTRef emptySeqInfo
    forM_ seq $ (modifySTRef cseq) . countSeq
    readSTRef cseq
```

StateT Monad Transformer

- StateT SeqInfo IO ()

```
countNuc5 :: String  -> IO SeqInfo
countNuc5 seq = do
    (_, cseq) <- runStateT (calc seq) emptySeqInfo
    return cseq
    where calc :: String -> StateT SeqInfo IO ()
          calc (x:xs) = do
              w <- get
              put $ countSeq x w
              calc xs
          calc [] = return ()
```

Interactive with StateT Monad

```
countNuc6 cseq = runStateT calc cseq
    where calc :: StateT SeqInfo IO ()
          calc = do
              n <- io $ hGetChar stdin
              if n == '\n' then return ()
              else do
                  w <- get
                  let w1 = countSeq n w
                  put w1
                  let (cA, cT, cG, cC) = w1
                  io $ when (w1 /= w) $
                  putStrLn (n : " (A:" ++ show cA ++ " T:"
                      ++ show cT ++ " G:"
                      ++ show cG ++ " C:"
                      ++ show cC ++ ")")
                  calc
          io = liftIO
```

IORef/MVar

- IORef is not a ‘computation’ to be run – it is just a box holding a simple value which may be used within IO in fairly arbitrary ways.
- IORef and MVar
  1. Concurrency: IORef use atomicModifyIORef, MVar uses a more general approach.
  2. MVar provides empty and fill two states, use for synchronization.

IORef

- API: newIORef, modifyIORef.

```
modifyIORef :: IORef a -> (a -> a) -> IO ()

countSeqIORef :: Char -> IORef SeqInfo -> IO ()
countSeqIORef n cseq = modifyIORef cseq $ countSeq n

countNuc3 :: String -> IORef SeqInfo -> IO SeqInfo
countNuc3 seq cseq = do
    mapM_ (flip countSeqIORef $ cseq) seq
    return =<< readIORef cseq
```

MVar

- API: newMVar, takeMVar, putMVar, modifyMVar

```
modifyMVar :: MVar a -> (a -> IO a) -> IO ()

countSeqMVar :: Char -> MVar SeqInfo -> IO ()
countSeqMVar n cseq = takeMVar cseq >>= (putMVar cseq)

countNuc4 :: String -> MVar SeqInfo -> IO SeqInfo
countNuc4 seq cseq = do
    mapM_ ((flip countSeqMVar) cseq) seq
    return =<< readMVar cseq
```

Network server

```
countSeqHandler :: MVar SeqInfo -> HandlerFunc
countSeqHandler mcseq addr msg = do
    (cA, cT, cG, cC) <- readMVar mcseq
    mapM_ ((flip countSeqMVar) mcseq) msg

main :: IO ()
main = do
    mcseq <- newMVar emptySeqInfo
    serveStub "11111" $ countSeqHandler mcseq
```

Summary

- State Monad and ST Monad: State provides a cleanest solution.
- StateT Monad Transformer: Interactive, storing global state in game
- IORef/MVar/TVar(STM, not covered): Mutable state which may be passed around and interacted with in controlled ways by IO code, MVar (faster), STM/TVar(slower)

Benchmark: no Option

- CountSeq/foldl'
  - mean: 8.367447 ms, lb 8.218304 ms, ub 8.597044 ms, ci 0.950
- CountSeq/State Monad
  - mean: 29.00443 ns, lb 28.40861 ns, ub 30.51695 ns, ci 0.950
- CountSeq/ST Monad
  - mean: 16.29765 ms, lb 16.01618 ms, ub 16.78243 ms, ci 0.950
- CountSeq/IORef
  - mean: 14.39716 ms, lb 13.95155 ms, ub 15.11095 ms, ci 0.950
- CountSeq/MVar
  - mean: 18.46335 ms, lb 18.09122 ms, ub 18.99158 ms, ci 0.950
- CountSeq/StateT
  - mean: 22.35278 ms, lb 21.90441 ms, ub 23.02252 ms, ci 0.950

Benchmark: -O

- CountSeq/foldl'
  - mean: 738.3003 us, lb 715.5623 us, ub 763.9446 us, ci 0.950
- CountSeq/State Monad
  - mean: 24.28070 ns, lb 23.67202 ns, ub 24.98461 ns, ci 0.950
- CountSeq/ST Monad
  - mean: 13.34422 ms, lb 12.95808 ms, ub 13.84491 ms, ci 0.950
- CountSeq/IORef
  - mean: 10.47555 ms, lb 10.22925 ms, ub 10.75831 ms, ci 0.950
- CountSeq/MVar
  - mean: 10.67125 ms, lb 10.37612 ms, ub 11.02434 ms, ci 0.950
- CountSeq/StateT
  - mean: 10.57837 ms, lb 10.28747 ms, ub 10.90067 ms, ci 0.950

Benchmark: -O3

- Build with -O3
- CountSeq/foldl'
  - mean: 499.2094 us, lb 487.7534 us, ub 512.6872 us, ci 0.950
- CountSeq/State Monad
  - mean: 26.12395 ns, lb 25.36894 ns, ub 26.92742 ns, ci 0.950
- CountSeq/ST Monad
  - mean: 10.44361 ms, lb 10.16934 ms, ub 10.74252 ms, ci 0.950
- CountSeq/IORef
  - mean: 10.73348 ms, lb 10.44679 ms, ub 11.05740 ms, ci 0.950
- CountSeq/MVar
  - mean: 11.03261 ms, lb 10.72550 ms, ub 11.37463 ms, ci 0.950
- CountSeq/StateT
  - mean: 10.63487 ms, lb 10.37687 ms, ub 10.92361 ms, ci 0.950