Mutable variables

A more thorough tutorial on STM is available as a Github repo.

Section exercise: write your own `TMVar` implementation.

Use cases:

- Communication among threads
- Let values survive an exception in a `StateT`
- Ugly hacks
- Inherently mutable algorithms (usually for performance)

IORef

Basic usage:

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script --optimize
import Data.IORef
import Control.Concurrent.Async (mapConcurrently_)

main :: IO ()
main = do
  ref <- newIORef (0 :: Int)
  modifyIORef ref (+ 1)
  readIORef ref >>= print
  writeIORef ref 2
  readIORef ref >>= print

  -- race condition
  let raceIncr = modifyIORef' ref (+ 1)
  writeIORef ref 0
  mapConcurrently_ id $ replicate 10000 raceIncr
  readIORef ref >>= print

  -- no race condition
  let noRaceIncr = atomicModifyIORef' ref (\x -> (x + 1, ()))
  writeIORef ref 0
  mapConcurrently_ id $ replicate 10000 noRaceIncr
  readIORef ref >>= print
```

To trigger the race condition, run like this:

```
 --resolver lts-12.21 exec -- ghc -O2 -threaded -with-rtsopts=-N foo.hs && ./foo
```

Survive exceptions

```haskell
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}
import Data.IORef
import UnliftIO (MonadUnliftIO, SomeException, tryAny)
import Control.Monad.Reader
import Control.Monad.State.Class
import System.Random (randomRIO)

newtype StateRefT s m a = StateRefT (ReaderT (IORef s) m a)
  deriving (Functor, Applicative, Monad, MonadIO, MonadTrans)

instance MonadIO m => MonadState s (StateRefT s m) where
  get = StateRefT $ ReaderT $ liftIO . readIORef
  put x = StateRefT $ ReaderT $ \ref -> liftIO $ writeIORef ref $! x

runStateRefT
  :: MonadUnliftIO m
  => StateRefT s m a
  -> s
  -> m (s, Either SomeException a)
runStateRefT (StateRefT (ReaderT f)) s = do
  ref <- liftIO $ newIORef s
  ea <- tryAny $ f ref
  s' <- liftIO $ readIORef ref
  return (s', ea)

main :: IO ()
main = runStateRefT inner 0 >>= print

inner :: StateRefT Int IO ()
inner =
    replicateM_ 10000 go
  where
    go = do
      res <- liftIO $ randomRIO (1, 100)
      if res == (100 :: Int)
        then error "Got 100"
        else modify (+ 1)
```

Memory usage

Let's calculate fibs (ugh).

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}
import Control.Monad
import Data.IORef

main :: IO ()
main = do
  fib1 <- newIORef (0 :: Int)
  fib2 <- newIORef (1 :: Int)

  -- we're gonna overflow, just ignore that
  replicateM_ 1000000 $ do
    x <- readIORef fib1
    y <- readIORef fib2
    writeIORef fib1 y
    writeIORef fib2 $! x + y

  readIORef fib2 >>= print
```

Run it with:

```
stack exec -- ghc foo.hs -O2 && ./foo +RTS -s
```

- `44,384 bytes maximum residency (1 sample(s))` yay!
- However...
- `16,051,920 bytes allocated in the heap`
- Linearly increases too

Problem: `IORef` always stores heap objects, resulting in GC pressure and pointer indirection. Mutable vectors to the rescue!

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}
import Control.Monad
import qualified Data.Vector.Unboxed.Mutable as V

main :: IO ()
main = do
  fib1 <- V.replicate 1 (0 :: Int)
  fib2 <- V.replicate 1 (1 :: Int)

  -- we're gonna overflow, just ignore that
  replicateM_ 1000000 $ do
    x <- V.unsafeRead fib1 0
    y <- V.unsafeRead fib2 0
    V.unsafeWrite fib1 0 y
    V.unsafeWrite fib2 0 $! x + y

  V.unsafeRead fib2 0 >>= print
```

- Much better: `51,936 bytes allocated in the heap`
- But that interface sucks
- rio to the rescue!

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE NoImplicitPrelude #-}
import RIO
import Prelude (print)

main :: IO ()
main = do
  fib1 <- newURef (0 :: Int)
  fib2 <- newURef 1

  -- we're gonna overflow, just ignore that
  replicateM_ 1000000 $ do
    x <- readURef fib1
    y <- readURef fib2
    writeURef fib1 y
    writeURef fib2 (x + y)

  readURef fib2 >>= print
```

- Explicit type signatures needed
- Oh yeah, `$!` isn't needed any more
- No `atomic` operations on unboxed arrays

Concurrency

- `atomicModifyIORef` works well for a single variable
- Cannot lock
- If you got two variables, you're gonna have a bad time

Onward and downward!

MVar

- Empty or full
- `take` and `put` block if var is empty or full, respectively
- Useful combinators like `modifyMVar`
- We can perform I/O while holding the variable
- Not great for multiple variables, we'll see STM next

```haskell
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE NoImplicitPrelude #-}
import RIO
import Prelude (print)
import System.Random (randomRIO)

main :: IO ()
main = do
    var <- newMVar (0 :: Int)
    replicateConcurrently_ 1000 (inner var)
    takeMVar var >>= print
  where
    inner var = modifyMVar_ var $ \val -> do
      -- I'm the only thread currently running. I could play around
      -- with some shared resource like a file
      x <- randomRIO (1, 10)
      return $! val + x
```

Also worth noting: `tryPutMVar` and `tryTakeMVar`.

Baton

Common pattern: send a notification between threads.

```haskell
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE OverloadedStrings #-}
import RIO
import Network.Wai
import Network.Wai.Handler.Warp
import Network.HTTP.Types
import Prelude (putStrLn, getLine)

app :: Application
app _req send = send $ responseBuilder status200 [] "Hello World"

main :: IO ()
main = do
    baton <- newEmptyMVar
    race_ (warp baton) (prompt baton)
  where
    warp baton = runSettings
      (setBeforeMainLoop (putMVar baton ()) defaultSettings)
      app
    prompt baton = do
      putStrLn "Waiting for Warp to be ready..."
      takeMVar baton
      putStrLn "Warp is now ready, type 'quit' to exit"
      fix $ \loop -> do
        line <- getLine
        if line == "quit"
          then putStrLn "Goodbye!"
          else putStrLn "I didn't get that, try again" >> loop
```

Software Transactional Memory

- Atomic transactions
- Trivially handles multiple variables
- No side effects allowed
- Lots of helper data structures
- Recommendation: make this your default for concurrent apps

Basics

STM comes with a few basic types and operations, and builds a rich ecosystem from them.

- `STM` is a monad in which all STM actions take place. It allows actions which read from and write to `TVar`s, but not other side effects (like writing to a file) which cannot be rolled back.
- `atomically :: STM a -> IO a` runs a block of STM actions atomically. Either everything succeeds, or nothing does. Since the resulting changes to mutable variables are visible to the entire program, it must be run from `IO`.
- `TVar` is a mutable variable, which can hold any data type.
- A standard, expected set of `TVar` creation and modification functions: `newTVar`, `readTVar`, and `writeTVar`.

These types and functions, along with many more, are exposed from the `Control.Concurrent.STM` module in the stm package.

EXERCISE Fill out the implementation of the following program so that it gives the output provided below.

```haskell
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
import Control.Concurrent.STM
import Control.Monad (replicateM_)

makeCounter :: IO (IO Int)
makeCounter = do
  var <- newTVarIO 1
  return undefined

main :: IO ()
main = do
  counter <- makeCounter
  replicateM_ 10 $ counter >>= print
```

Should print:

```
1
2
3
4
5
6
7
8
9
10
```

Failure, retry, and alternative

One of the most powerful concepts in STM is the ability to retry. As a motivating example: let's say we have two `TVar`s, representing the bank accounts for Alice and Bob. Alice makes $5 every second (pretty nice!), and wants to give Bob $20. Our code might initially look like:

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
import Control.Concurrent
import Control.Concurrent.STM
import Control.Monad (forever)
import Say

main :: IO ()
main = do
  aliceVar <- newTVarIO 0
  bobVar <- newTVarIO 0

  _ <- forkIO $ payAlice aliceVar

  atomically $ do
    currentAlice <- readTVar aliceVar
    writeTVar aliceVar (currentAlice - 20)
    currentBob <- readTVar bobVar
    writeTVar bobVar (currentBob + 20)

  finalAlice <- atomically $ readTVar aliceVar
  finalBob <- atomically $ readTVar bobVar

  sayString $ "Final Alice: " ++ show finalAlice
  sayString $ "Final Bob: " ++ show finalBob

payAlice :: TVar Int -> IO ()
payAlice aliceVar = forever $ do
  threadDelay 1000000
  atomically $ do
    current <- readTVar aliceVar
    writeTVar aliceVar (current + 5)
  sayString "Paid Alice"
```

There are no race conditions, thanks to STM. But this program is still buggy, at least at the logic bug level. The issue is that we allow Alice to give money she doesn't have! Let's look at our output:

```
Final Alice: -20
Final Bob: 20
```

Instead, we want to check that Alice's balance is at least $20 before we let her transfer the money. In order to do this, we just need to use one new helper function:

```
check :: Bool -> STM ()
check b = if b then return () else retry
```

As the implementation indicates, this function will do nothing if `b` is true, but will *retry* if it's false. This is the second bit of magic in STM. Since STM needs to track all of the variables it has looked at in order to handle transactions, it knows exactly what led to its current state. If you call `retry`, you're saying to STM, "I don't like this result. Run me again when one of the variables I looked at changed, and I'll decide if things are OK now."

EXERCISE Use the `check` function to modify the program above so Alice's bank balance is never negative. The output should look something like this:

```
Paid Alice
Paid Alice
Paid Alice
Paid Alice
Final Alice: 0
Final Bob: 20
```

On top of this retry behavior, STM also implements an `Alternative` instance which allows us to try a number of different transactions until one of them succeeds. For example, let's say both Alice and Bob are trying to give Charlie $20. We can wait to see who sends the money first.

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
import Control.Applicative ((<|>))
import Control.Concurrent
import Control.Concurrent.STM
import Control.Monad (forever, void)
import Say

main :: IO ()
main = do
  aliceVar <- newTVarIO 0
  bobVar <- newTVarIO 0
  charlieVar <- newTVarIO 0

  payThread aliceVar 1000000 5
  payThread bobVar   1500000 8

  atomically $ transfer 20 aliceVar charlieVar
           <|> transfer 20 bobVar   charlieVar

  finalAlice <- atomically $ readTVar aliceVar
  finalBob <- atomically $ readTVar bobVar
  finalCharlie <- atomically $ readTVar charlieVar

  sayString $ "Final Alice: " ++ show finalAlice
  sayString $ "Final Bob: " ++ show finalBob
  sayString $ "Final Charlie: " ++ show finalCharlie

payThread :: TVar Int -> Int -> Int -> IO ()
payThread var interval amount = void $ forkIO $ forever $ do
  threadDelay interval
  atomically $ do
    current <- readTVar var
    writeTVar var (current + amount)

transfer :: Int -> TVar Int -> TVar Int -> STM ()
transfer amount fromVar toVar = do
  currentFrom <- readTVar fromVar
  check (currentFrom >= amount)
  writeTVar fromVar (currentFrom - amount)
  currentTo <- readTVar toVar
  writeTVar toVar (currentTo + amount)
```

The runtime system will try to transfer money from Alice. If that fails, it will try to transfer money from Bob. If that fails, it will wait until either Alice or Bob's account balance changes, and then try again.

QUESTION Would it be possible to get the desired behavior if `transfer` called `atomically` itself and returned `IO ()` instead of `STM ()`? If so, write the program. If not, why not?

Other helper functions

You may have already discovered in the previous exercises that there are a number of other helper functions to work with `TVar`s. One example is:

```
modifyTVar :: TVar a -> (a -> a) -> STM ()
```

If you're accustomed to dealing with thread-safe code, you may expect this to use a special implementation internally to perform some locking. But remember: with STM, locking is unnecessary. Instead, this is nothing more than a convenience function, with the very simple implementation:

```
modifyTVar :: TVar a -> (a -> a) -> STM ()
modifyTVar var f = do
    x <- readTVar var
    writeTVar var (f x)
```

EXERCISE This modify function is lazy. Can you change it to be strict?

Why newTVarIO?

One of the helper functions available is:

```
newTVarIO :: a -> IO (TVar a)
```

It may seem like this should have the obvious implementation of `atomically . newTVar`. However, this implementation would be bad for two reasons. The first is that it's inefficient: it requires all of the machinery for running a transaction, when by its nature we know that creating a new `TVar` will never fail.

The second is more subtle. Let's say we're going to follow the common (albeit arguably evil) practice of creating a global mutable variable. You'll end up with some code that looks like this:

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
import Control.Concurrent.STM
import Control.Monad (replicateM_)
import System.IO.Unsafe (unsafePerformIO)

callCount :: TVar Int
callCount = unsafePerformIO $ atomically $ newTVar 0
{-# NOINLINE callCount #-}

someFunction :: IO ()
someFunction = do
  count <- atomically $ do
    modifyTVar callCount (+ 1)
    readTVar callCount
  putStrLn $ "someFunction call count: " ++ show count

main :: IO ()
main = replicateM_ 10 someFunction
```

Besides the nauseating call to `unsafePerformIO`, everything looks fine. Unfortunately, running this application fails:

```
Main.hs: Control.Concurrent.STM.atomically was nested
```

The issue is that, in order to properly implement STM, you cannot embed one call to `atomically` inside another call to `atomically`. "But wait," you exclaim, "the types prevent that from ever happening! You can't run an `IO` action inside an `STM` action!" The problem is that `unsafePerformIO` let us do just this. `callCount` starts as a thunk which, when evaluated, calls `atomically`. And the first time it's evaluated is at `modifyTVar callCount (+ 1)`, which is *also* inside `atomically`!

For these two reasons, the `newTVarIO` function exists. This isn't a promotion of global variables, but simply an explanation: if you're going to use them, do them correctly.

There's also a `readTVarIO` function available. This one is present purely for performance reasons, as reading a `TVar` is always a non-failing operation.

EXERCISE Fix up the code above so that it doesn't throw an exception.

QUESTION Is it possible to use `readTVarIO` in this program? Is it safe? Would it still be safe if we introduced some concurrency?

Other variable types

`TVar`s are the core variable type in STM. However, as a convenient, the `stm` library provides a number of other variable types built on top of it. We'll demonstrate a few here.

TMVars

Channels and queues

There are three related variable types in `stm`:

- `TChan` is an unbounded FIFO channel. You write things to one end, and read them at the other end.
- `TQueue` is just like a `TChan`, but it doesn't support a concept known as *broadcast*, with multiple readers for a single writer. In exchange, `TQueue`s are faster. Takeaway: unless you specific need broadcast (which in my experience is a rare occurrence), prefer queues.
- `TBQueue` is like a `TQueue`, but is also bounded. If more than the given amount of values are present in the queue already, further

writes will `retry` until the queue has been drained.

In addition to these types, the `stm-chans` library provides a number of additional channel and queue types, including variants which can be closed. Let's use one of these to explore the basic API and bit, and implement a concurrent URL downloader in the process.

NOTE This example takes advantage of the wonderful async library, which goes hand-in-hand with STM. Once you've finished this tutorial, it's strongly advised to go and read about async to get the rest of the story with concurrency in Haskell. We're also using the http-conduit library for HTTP requests.

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
import Control.Concurrent.Async
import Control.Concurrent.STM
import Control.Concurrent.STM.TBMQueue
import Control.Exception (finally)
import Control.Monad (forever, void)
import qualified Data.ByteString.Lazy as BL
import Data.Foldable (for_)
import Network.HTTP.Simple

main :: IO ()
main = do
  queue <- newTBMQueueIO 16
  concurrently_
    (fillQueue queue `finally` atomically (closeTBMQueue queue))
    (replicateConcurrently_ 8 (drainQueue queue))

fillQueue :: TBMQueue (String, String) -> IO ()
fillQueue queue = do
  contents <- getContents
  for_ (lines contents) $ \line ->
    case words line of
      [url, file] -> atomically $ writeTBMQueue queue (url, file)
      _ -> error $ "Invalid line: " ++ show line

drainQueue :: TBMQueue (String, String) -> IO ()
drainQueue queue =
    loop
  where
    loop = do
      mnext <- atomically $ readTBMQueue queue
      case mnext of
        Nothing -> return ()
        Just (url, file) -> do
          req <- parseRequest url
          res <- httpLBS req
          BL.writeFile file $ getResponseBody res
```

Exceptions

You can throw and catch exceptions inside an STM block if desired. The semantics are the same as catching and handling exception inside `IO` itself. Inside of `throwIO` and `catch`, you just use `throwSTM` and `catchSTM`.

The join trick

There's a nifty little trick you can use when writing STM code. A common pattern is to want to perform some `IO` at the end of a transaction. Since you can't run it inside the transaction itself, you instead run it right after the transaction. For example:

```haskell
addFunds :: TVar Int -> Int -> IO ()
addFunds var amt = do
  new <- atomically $ do
    orig <- readTVar var
    let new = orig + amt
    writeTVar var new
    return new
  putStrLn $ "New amount: " ++ show new
```

Having to break up your logic like that feels wrong, so instead of simply returning the `new` value, we can instead return an `IO` action to be run after the block:

```haskell
addFunds :: TVar Int -> Int -> IO ()
addFunds var amt = do
  action <- atomically $ do
    orig <- readTVar var
    let new = orig + amt
    writeTVar var new
    return $ putStrLn $ "New amount: " ++ show new
  action
```

And then we can use the `join` function to clean things up a bit further:

```haskell
addFunds :: TVar Int -> Int -> IO ()
addFunds var amt = join $ atomically $ do
  orig <- readTVar var
  let new = orig + amt
  writeTVar var new
  return $ putStrLn $ "New amount: " ++ show new
```

```haskell
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE OverloadedStrings #-}
import RIO
import Say

main :: IO ()
main = do
  seller    <- newTVarIO (0 :: Int)
  buyer     <- newTVarIO (100 :: Int)
  purchases <- newTVarIO (0 :: Int)
  taxes     <- newTVarIO (0 :: Int)
  let makePurchase = join $ atomically $ do
        buyer' <- readTVar buyer
        if buyer' < 10
          then return $ say "Not enough money to make purchase"
          else do
            modifyTVar' buyer (subtract 10)
            modifyTVar' seller (+ 9)
            modifyTVar' taxes (+ 1)
            modifyTVar' purchases (+ 1)
            return $ say "Purchase successful"
  replicateConcurrently_  20 makePurchase
```

Exercises

- Rewrite Warp example with STM
  - Use a `TMVar` baton
  - Use `TVar` and `check`
- Implement your own `TMVar`