1. Remember: Core STM Concepts and Concurrency Issues

Define `TVar` and Basic `STM` Operations

```
import Control.Concurrent.MVar
import Control.Concurrent

-- Define a transactional variable (TVar)
data TVar a = TVar (MVar a)

-- Function to create a new TVar
newTVar :: a -> IO (TVar a)
newTVar val = TVar <$> newMVar val

-- Function to read from a TVar
readTVar :: TVar a -> IO a
readTVar (TVar mvar) = do
                val <- readMVar mvar
                threadDelay 5000000 -- Add a delay of 5 second
                return val

-- Function to write to a TVar
writeTVar :: TVar a -> a -> IO ()
writeTVar (TVar mvar) val = modifyMVar_ mvar (\_ -> return val)
```

Identifying concurrency issues: This simple version of `TVar` does not yet solve concurrency problems like race conditions or deadlocks. It uses `MVar` for thread safety but does not handle transaction rollbacks or retries.
Atomicity, Consistency, Isolation: The next steps will aim to build the transaction mechanism to achieve atomicity and isolation by running multiple operations as a single unit (the core of STM).

As an experiment, we can proceed as follows:

1. Define thread variable `tv` with a default value.
2. Read and double check the value
3. Have a `readTVar` func running in background
4. Attempt to `writeTVar` during the background execution
5. Read values to double check.

As a result, everything plays out as expected, the function reads the value before the value gets overwritten.

```
ghci> :l main1.hs
[1 of 2] Compiling Main             ( main1.hs, interpreted )
Ok, one module loaded.
ghci> tv <- newTVar 0
ghci> readTVar tv >>= print
0
ghci> forkIO (readTVar tv >>= print)
ThreadId 868
ghci> writeTVar tv 100
ghci> 0

ghci> readTVar tv >>= print
100
ghci>
```

2. Understand: STM for Preventing Concurrency Issues

Next, I implement a very basic version of STM to group operations on `TVar`s within an atomic block. This will help prevent inconsistent states when multiple threads access shared resources concurrently.

Implement a Simple STM Monad

```
import Control.Exception (try, SomeException)
import Control.Concurrent.MVar
import Control.Concurrent

-- Define a transactional variable (TVar)
data TVar a = TVar (MVar a)

-- Function to create a new TVar
newTVar :: a -> IO (TVar a)
newTVar val = TVar <$> newMVar val

-- Function to read from a TVar
readTVar :: TVar a -> IO a
readTVar (TVar mvar) = do
                val <- readMVar mvar
                threadDelay 15000000 -- Add a delay of 5 second
```

```haskell
                return val

-- Function to write to a TVar
writeTVar :: TVar a -> a -> IO ()
writeTVar (TVar mvar) val = modifyMVar_ mvar (\_ -> return val)

-- Stage 2 update
-- Define a simple STM monad
newtype STM a = STM { runSTM :: IO a }

-- The atomically function to execute STM actions
atomically :: STM a -> IO a
atomically (STM action) = do
    -- Wrap action in exception handling to simulate a retry mechanism
    result <- try action
    case result of
      Right val -> return val
      Left (_ :: SomeException) -> atomically (STM action)  -- Retry on failure

-- Functions to lift TVar operations into STM
readTVarSTM :: TVar a -> STM a
readTVarSTM tvar = STM $ readTVar tvar

writeTVarSTM :: TVar a -> a -> STM ()
writeTVarSTM tvar val = STM $ writeTVar tvar val
```

Here, I introduced the `STM` monad, which wraps IO operations on `TVar`s. The `atomically` function executes `STM` transactions, and in case of an exception (such as a conflict in a real STM system), it retries the operation.

- **Concurrency issues and STM's retry mechanism**: We simulate the retry mechanism here by catching exceptions, but a real implementation would involve detecting conflicts between transactions.

We can repeat the same experiment as mentioned in the previous stage which yields the same expected result.

```
ghci> :l main2.hs
[1 of 2] Compiling Main             ( main2.hs, interpreted )
Ok, one module loaded.
ghci> tv <- newTVar 0
ghci> tv <- newTVar 1
ghci> atomically (readTVarSTM tvar) >>= print

<interactive>:53:25: error: Variable not in scope: tvar :: TVar a0
ghci> atomically (readTVarSTM tv) >>= print
1
ghci> forkIO (atomically (readTVarSTM tv) >>= print)
ThreadId 1566
ghci> atomically (writeTVarSTM tv 100)
ghci> 1

ghci>
ghci>
ghci> atomically (readTVarSTM tv) >>= print
100
```

3. Apply: Implement a Job Scheduler Using the Custom STM

Next, I built a simple job queue using the `STM` and `TVar` implementations which will process jobs concurrently while using STM to ensure safe access to shared data.

````
```

type JobQueue = TVar [String]

-- Submit a job to the queue
submitJobSTM :: JobQueue -> String -> STM ()
submitJobSTM queue job = do
    jobs <- readTVarSTM queue
    writeTVarSTM queue (jobs ++ [job])

-- Process a job from the queue
processJobSTM :: JobQueue -> STM (Maybe String)
processJobSTM queue = do
    jobs <- readTVarSTM queue
    case jobs of
      [] -> return Nothing
````
````

```
      (job:rest) -> do
        writeTVarSTM queue rest
        return (Just job)

-- Concurrently process jobs using custom STM
processJobsConcurrentlySTM :: JobQueue -> IO ()
processJobsConcurrentlySTM queue = atomically $ do
    mJob <- processJobSTM queue
    case mJob of
      Just job -> STM $ putStrLn ("Processing job: " ++ job)
      Nothing  -> return ()
```

This job scheduler submits and processes jobs using the custom `TVar` and `STM`. It ensures safe concurrent access by grouping `readTVar` and `writeTVar` operations into atomic transactions

To test the job queue, I also added a test case to analyze how it works.

```
-- Test case to test concurrency in the job queue
testConcurrentJobProcessing :: IO ()
testConcurrentJobProcessing = do
    -- Step 1: Create an empty job queue
    queue <- newTVar []

    -- Step 2: Spawn multiple threads to concurrently submit jobs
    let numJobs = 10  -- Total number of jobs to submit
    putStrLn "Submitting jobs concurrently..."
    replicateM_ 10 $ forkIO $ replicateM_ (numJobs `div` 10) $ do
        atomically $ submitJobSTM queue "TestJob"

    -- Step 3: Spawn multiple threads to process jobs concurrently
    putStrLn "Processing jobs concurrently..."
    replicateM_ 5 $ forkIO $ replicateM_ (numJobs `div` 5) $
processJobsConcurrentlySTM queue

    -- Step 4: Allow time for job submission and processing
    threadDelay 2000000  -- Wait for 2 seconds to allow all jobs to be
processed

    -- Step 5: Check if all jobs were processed
```

```
    jobsRemaining <- readTVar queue
    putStrLn $ "Remaining jobs: " ++ show jobsRemaining

    -- Since we've submitted `numJobs` jobs, the queue should be empty after all are processed
    when (null jobsRemaining) $
        putStrLn "All jobs processed. PASS"
    when (not (null jobsRemaining)) $
        putStrLn "Some jobs were not processed. FAIL"
```

And here is the result of it:

```
ghci> testConcurrentJobProcessing
Submitting jobs concurrently...
Submitting jobs concurrently...
Processing jobs concurrently...
Processing jobs concurrently...
PrPPoPrPrcroroeoc属cscecesesesisssssnsisiginin ngngjg g o j jbjojo:obob b:b:T: : e T TsTeTetesesJststotJtJ
bJoJo
obobb
b


Remaining jobs: []
All jobs processed. PASS
ghci>
```

4. Analyze: Implementing and Benchmarking Concurrency

To analyze the performance of my custom implementation, I implemented two versions of the concurrency code, one with custom `STM` and one with `MVar`.

The main difference between is how the JobQueue is instantiated

CustomSTM	MVar
`type JobQueue = TVar [String]`	`type JobQueue = MVar [String]`

After a couple of experiments, there is no significant difference between the two implementations. As expected, `CustomSTM` performs worse in time (by 0.2%).

Here are the results of the test (`CustomSTM` above, `MVar` below):

```
ghci> :l main4tvar.hs
[1 of 2] Compiling Main             ( main4tvar.hs, interpreted )

main4tvar.hs:49:5: warning: [-Wnoncanonical-monad-instances]
    Noncanonical `return' definition detected
    in the instance declaration for `Monad STM'.
    `return' will eventually be removed in favour of `pure'
    Either remove definition for `return' (recommended) or define as `return = pure'
    See also: https://gitlab.haskell.org/ghc/ghc/-/wikis/proposal/monad-of-no-return
   |
49 |     return x = STM (return x)
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^
Ok, one module loaded.
ghci> testCustomSTMJobProcessing
Submitting jobs concurrently...
All jobs processed. PASS
Elapsed time: 0.2102979s
```

```
ghci> :l main4mvar.hs
[1 of 2] Compiling Main             ( main4mvar.hs, interpreted )
Ok, one module loaded.
ghci> testConcurrentJobProcessing
Submitting jobs concurrently...
All jobs processed. PASS
Elapsed time: 0.2098653s
```