**Reading**

1. Beautiful Concurrency (see the Reading links for the Software Transactional Memory lecture on CourseWare)
2. The Transactional Memory / Garbage Collection Analogy (see the Reading links for the Software Transactional Memory lecture on CourseWare)
3. Tutorial on Parallel and Concurrent Programming in Haskell (see the Reading links for the Parallelism in Haskell lecture on CourseWare)

**Problems**

1. ................................. Software Transactional Memory

   Peter is starting his new job at Initech. His first task is to take some atomic Haskell code snippets and convert them to atomic Java code. One of the snippets he comes across is the following:

   ```haskell
   atomic (do { readVar x; 
                writeVar x })
   ```

   Peter writes his Java code and submits it to his boss Bill. Bill is concerned that the code is not getting called enough, so he lets Peter know that it would be terrific if he could keep track of how many times the code snippet is actually executed. Peter finally comes up with the following java code:

   ```java
   public class AtomicDoIt {
       public CallCounter callCounter;
       public AtomicDoIt() {
           callCounter = new CallCounter(0);
       }
       public synchronized void doIt(int x) {
           callCounter.increment(); // keep track of how many times doIt is executed
           readVar(x);
           writeVar(x);
       }
   }
   ```

   (a) Why might this code not work as Bill and Peter expect it to?
   (b) Why is it safe to allow side effects on transactional variables?
   (c) What makes Haskell a good fit for STM?
MVars using STM in Haskell

In our IO Monad lectures we encountered IORefs – Haskell mutable references. Though IORefs are widely used in sequential Haskell programs, in the presence of concurrency IORefs are of little use: we cannot control when a thread writes to an IORef. However, we can use Haskell MVars. An MVar, of type \( \text{MVar} \ a \), is mutable location that can be in one of two states; an MVar can be empty, or it can be full with a value of type \( a \).

Similar to IORefs, we have several combinators to work with MVars. The core function \( \text{putMVar} \) is used to fill an empty MVar (if the MVar is not empty, \( \text{putMVar} \) blocks). Dually, \( \text{takeMVar} \) is used to read (and clear) a filled MVar (if the MVar is empty, \( \text{takeMVar} \) blocks). Among other uses, MVars are typically used for synchronization, and as typed one-place communication-channels (\( \text{putMVar} \) corresponding to a \textit{send}; \( \text{takeMVar} \) corresponding to a \textit{receive}). In this problem we will implement MVars using STM, and use our MVar library to implement a toy example. We provide two source files, \texttt{MVar.hs} and \texttt{Main.hs}. You will answer parts (a)-(c) of this question directly in \texttt{MVar.hs} and part (d) in \texttt{Main.hs}. Submit your modified versions on CourseWare.

Since an MVar \( a \) can hold a value of type \( a \) or be empty, we can represent an MVar as a TVar holding a value of type \( \text{Maybe} \ a \):

\[
\text{type MVar a = TVar (Maybe a)}
\]

Following, we define two constructors for MVars. The first, \( \text{newEmptyMVar} \), creates an empty MVar. While the second, \( \text{newMVar} \), creates an MVar filled with the given value. The following code implements this:

\[
\begin{align*}
\text{newEmptyMVar} & \text{ :: STM (MVar a)} \\
\text{newEmptyMVar} & = \text{newTVar Nothing} \\
\text{newMVar} & \text{ :: } a \rightarrow \text{STM (MVar a)} \\
\text{newMVar} \ x & = \text{newTVar (Just x)}
\end{align*}
\]

(a) Consider the partial implementation of \( \text{takeMVar} \):

\[
\begin{align*}
\text{takeMVar} & \text{ :: MVar a } \rightarrow \text{ STM a} \\
\text{takeMVar} \ mv & = \\
& \text{ do } v \gets \text{readTVar } mv \\
& \text{ case } v \text{ of} \\
& \quad \text{Nothing } \rightarrow \quad \text{__________________________} \\
& \quad \text{Just val } \rightarrow \quad \text{__________________________} \\
& \quad \text{__________________________}
\end{align*}
\]

As previously mentioned, \( \text{takeMVar} \) has the semantics that if the MVar \( mv \) is empty the function blocks until the MVar is filled by another thread. Once the MVar is filled, \( \text{takeMVar} \) reads the value (emptying the “slot”) and returns the read value. Complete the above implementation in \texttt{MVar.hs}.

(b) Similarly, consider the partial implementation of \( \text{putMVar} \):

\[
\begin{align*}
\text{putMVar} & \text{ :: MVar a } \rightarrow \ a \rightarrow \text{ STM ()} \\
\text{putMVar} \ mv \ newval & = \\
& \text{ do } v \gets \text{readTVar } mv \\
& \text{ case } v \text{ of} \\
& \quad \text{Nothing } \rightarrow \quad \text{__________________________} \\
& \quad \text{Just val } \rightarrow \quad \text{__________________________} \\
& \quad \text{__________________________}
\end{align*}
\]
The dual of \texttt{takeMVar}, \texttt{putMVar} has the semantics that if the MVar \(mv\) is not empty the function blocks until the MVar is emptied by another thread. Once the MVar is empty, \texttt{putMVar} writes the value \(newval\) into the “slot”. Complete the above implementation in MVar.hs.

(c) Blocking until an MVar is in the correct state (empty or full) is not always a desired effect; in many applications blocking is not permissible. Hence, we implement non-blocking versions of \texttt{takeMVar} and \texttt{putMVar} called \texttt{tryTakeMVar} and \texttt{tryPutMVar}, respectively. A partial implementation is given below.

\begin{align*}
\text{tryTakeMVar :: MVar a} &\rightarrow \text{STM (Maybe a)} \\
\text{tryPutMVar :: MVar a} &\rightarrow \text{STM Bool}
\end{align*}

The \texttt{tryTakeMVar} function is equivalent to \texttt{takeMVar} except that, instead of blocking, it returns a Maybe-valued transaction indicating whether the operation was successful. Specifically, \texttt{tryTakeMVar} returns \texttt{Nothing} if the MVar \(mv\) is empty, or \texttt{Just a} if the MVar is full with value \(a\), leaving it empty.

The \texttt{tryPutMVar} function is equivalent to \texttt{putMVar} except that, instead of blocking, it returns a Boolean-valued transaction indicating whether the operation was successful. In other words, \texttt{tryPutMVar} takes an MVar \(mv\) and a new value \(newval\) and tries to put the new value into the MVar. If the attempt is successful, the function returns \texttt{True}; otherwise it returns \texttt{False}.

Complete the implementation of \texttt{tryTakeMVar} and \texttt{tryPutMVar} in Main.hs.

(d) Consider a program that uses three threads to atomically update a shared integral MVar (initially set to 0) until the value of the MVar reaches a specified limit (100 in our example). Each thread has the task of repeatedly trying to add their quantum to the MVar value. In this example, the the quantums are 3, 13 and 1. When the threads can no longer increment the value in the shared MVar, they terminate. The main thread waits for all the threads, reads the final value and prints it out. Below is the code for the main thread.

\begin{verbatim}
main = do
  mvar <- atomically (newTMVar 0) -- Create shared MVar
  let limit = 100
  t0 <- genTask mvar limit 3  -- Create new task with quantum 3
  t1 <- genTask mvar limit 13 -- Create new task with quantum 13
  t2 <- genTask mvar limit 1  -- Create new task with quantum 1
  waitForAll [t0,t1,t2] -- Wait for the threads to finish
  result <- atomically (takeTMVar mvar) -- Read final result
  putStrLn (“Final result: “ ++ (show result))
\end{verbatim}

Here, \texttt{genTask} creates an empty synchronization MVar (of type \texttt{MVar ()}), spawns a new thread which executes \texttt{task} with the given shared MVar \texttt{mvar}, limit, and quantum. Once the thread finishes, it uses \texttt{putMVar} to fill the synchronization MVar. The main thread uses \texttt{tryTakeMVar} in \texttt{waitForAll} to wait for all the threads to finish. We encourage you to read the code in Main.hs to understand the use of MVars for synchronization.

Given the shared MVar, the limit, and increment quantum, the \texttt{task} function uses the \texttt{inc} transaction to atomically increment the TMVar by the quantum. If the increment is successful, then the thread is not necessarily done and recursively calls the \texttt{task} function to repeat the process. However, if it can no longer increment the shared MVar, the thread finishes by returning the unit value.

The code for the increment transaction is given below.
inc :: TMVar Int -> Int -> Int -> STM Bool
inc mvar limit myQuantum =
do oldAmount <- takeTMVar mvar -- Read old value
let newAmount = myQuantum + oldAmount -- Add quantum
if newAmount > limit
then do putTMVar mvar oldAmount -- Leave old value
      return True -- Done
else do putTMVar mvar newAmount -- Change value
      return False -- Not done

A partial implementation of task is given below.
task :: TMVar Int -> Int -> Int -> IO ()
task mvar limit quantum =
do done <- ___________________________________________
if done
then return ()
else task mvar limit quantum

Complete the implementation of task in Main.hs. The function must use the inc transaction given above. You will know when your implementation works when your main function prints this out to the terminal as its final result:

Final result: 100.

(e) Explain what happens to the program if one of the calls to putMVar is omitted from inc.

3. .................................................. Software Transactional Memory

(a) Two different Haskell threads, \( T_s \) and \( T_l \), execute in parallel, sharing access to a common variable \( x \) initialized to 0. \( T_s \) repeatedly executes a task, \( t_{short} \), that increments \( x \) by one 10 times. \( T_l \) repeatedly executes a task, \( t_{long} \), that increments \( x \) by one 10,000 times. Both threads terminate when \( x \) reaches 1,000,000.
Consider two different synchronization scenarios between \( T_s \) and \( T_l \).
Scenario 1: the threads acquiring a lock before each execution of their respective task and then release the lock after each task completion.
Scenario 2: the threads wrap the tasks up in atomic transactions, with \( x \) as a TVar.
Explain the possible starvation in both scenarios. In Scenario 1, has \( T_s \) or \( T_l \) incremented \( x \) more times when both threads terminate? How about in Scenario 2? (Consider only the committed values of \( x \) for Scenario 2.) Explain both answers briefly.

(b) One advantage of using transactions over locking is that transactions have a “composable” property while locking does not. This problem asks you to create a transaction composed from other transactions.
Consider the following Haskell STM implementation of a bank account.

    type AccountH = TVar Int

    withdraw :: AccountH -> Int -> STM()
    withdraw acc amount = do bal <- readTVar acc
                                writeTVar acc (bal - amount)

    deposit :: AccountH -> Int -> STM()
    deposit acc amount = withdraw acc (-amount)

Write Haskell code to do an atomic transfer transaction composed from withdraw and deposit using the provided skeleton below. transfer should deduct amount from accl
and add it to acc2. Note that atomically was written as atomic in the lecture slides for brevity.

```haskell
transfer :: AccountH -> AccountH -> Int -> IO ()
transfer acc1 acc2 amount = atomically (do ______________________

______________________
)
```

(C) The concept of “Weak Atomicity” in transactional memory means that non-atomic blocks have the ability to observe uncommitted temporary state in the middle of an atomic block. Consider the following weakly-atomic version of the Account transfer function, written in a hypothetical version of Java supporting weak atomicity.

```java
void transfer_weak_atomic(AccountJ other, float amt) {
  weak_atomic {
    other.balance -= amt;
    this.balance += amt;
  }
}

int getSum(AccountJ other) {
  return other.balance + this.balance;
}
```

Suppose we have 2 instances of AccountJ: x and y, both initialized to have balance equal to 100. One thread executes y.transfer_weak_atomic(x, 1) and another thread executes y.getSum(x) in parallel. Assume there is no compiler re-ordering (inside getSum, other.balance is always read before this.balance, and inside transfer_weak_atomic, other.balance is always decremented before this.balance is incremented).

What possible values may be returned by y.getSum(x)? Explain briefly.

(d) Below is the Haskell implementation of getSum, using the AccountH Haskell type.

```haskell
getSum :: AccountH -> AccountH -> STM Int
getSum x y = do xtemp <- readTVar x
                 ytemp <- readTVar y
                 return (xtemp + ytemp)
```

How does the Haskell implementation prevent weak atomicity? More specifically, consider a main program coding the same execution scenario as in part (c): x and y are both of type AccountH and are initialized to 100. Then, transfer x y 1 runs in one thread while getSum x y runs in another parallel thread. Describe the mechanism(s) that prevent getSum x y from returning a result other than 200. (To fully answer this question, you need to consider the type of atomically, main, getSum, and transfer.)

4. .......................................................... Data Parallel Haskell

John Anderton has started his new job working for The Bureau. His first job is to help design an image recognition system to detect objects from numerous streams of video. His first task is to identify coffee mugs found in each video frame.

The Bureau has already developed a classifier function that will accurately determine if a coffee mug is within an image of size 100x100 pixels. You will help Mr. Anderton write the rest of the program, which will split each frame of video into a regular grid of 100x100 pixel subimages and check each one for a mug.

You will be given helper functions which will do a lot of the heavy lifting. You just have to put all the pieces together.
(a) The Bureau has a plentiful research budget for these types of projects, and is actively exploring two architectures for the image recognition pipeline. The first uses Data Parallel Haskell (as introduced in lecture). Help John write a flat data parallel Haskell program that will classify each of the images in parallel by filling in the blanks:

```haskell
type Image -- The data type for an image

classify :: Image -> Bool
-- Library function you can call that will accurately determine if
-- there is a mug in the image, which must be 100x100.

grid :: Image -> [Image]
-- Library function you can call that takes an image and returns a
-- list of 100x100 pixel images you check with the classify function.

-- When given a database of images, the detector must determine if
-- there is a mug present in each image.
detector :: [: Image :] -> [: Bool :]
detector images =
  ____ result | image <- images
    , let result = classifyAll (____________ image) ____
classifyAll :: ________ -> Bool
classifyAll subimages =
  orP ____ _______________ | subimage <- subimages ____
-- Hint: "or" is a library function with type [Bool] -> Bool
```

(b) In 2-3 sentences, describe how this program works. In your description, tell us exactly which function calls in this program are executed in parallel.

(c) The second architecture the Bureau is exploring attempts to use Data Parallel Haskell to write a nested data parallel program. While searching the Bureau’s code repository, John found a function fromList :: [a] -> [: a :] that he thinks might be useful in addition to the classify and grid functions shown above. Help John figure out how to write the rest of his program:

```haskell
fromList :: [a] -> [: a :]
-- Library function you can call that might be useful
detector :: [: Image :] -> [: Bool :]
detector images =
  ____ result | image <- images
    , let result = classifyAll (________ image)) ____
classifyAll :: ________ -> Bool
classifyAll subimages =
  orP _______ ____ || subimage <- subimages ____
-- Hint: "orP" is a data parallel function with type [: Bool :] -> Bool
```

(d) Briefly describe how the nested data parallel program is different from the flat data parallel program. Describe exactly which parts of this program are run in parallel that are not run in parallel in the previous version.

(e) The images in the database range from 1000x1000 pixels to 100x100 pixels. Thus it will take 100 calls to the function classify to check the largest image for a mug and one call to check
the smallest. Assume that both students run their code on a machine with 10 processors. A test database contains 1 maximum size image and 9 minimum size images.

For the flat parallel program, how many calls to classify does each processor make? For the nested parallel program, how many calls to classify does each processor make given the same test database?

(f) John’s boss Lamar offers an idea. He wants to take John’s program and change classifyAll so that as soon as one call to classify returns true, the function ends the rest of its computations and immediately returns true.

Currently, such short-circuited evaluation is not supported by Data Parallel Haskell. Briefly describe what the Data Parallel Haskell compiler and/or run-time system would have to do to support short-circuited evaluation.