Department of Information and Computer Science
Utrecht University

# INFOB3CC: STM (solutions)

Trevor L. McDonell

December 5, 2021

## Introduction

Use these tasks to practice the topics of the lectures. You may have to do some research to read up on terms or topics not (yet) covered in the lectures.

## Questions

1. Properties of software transactional memory.
   (a) What is the purpose of the function `atomically :: STM a -> IO a`?

   **Solution:** The `atomically` function executes the actions given in the first argument as a single atomic transaction. All of the actions defined in the atomic block appear to the rest of the program to take place as a single indivisible operation.

   (b) Explain why operations in software transactional memory are defined using the type `STM`.

   **Solution:** Because software transactional memory relies on the ability to abort and retry transactions when a conflict is detected, we must limit the operations inside the atomic block to actions which can be undone. For example, most IO actions such as printing to the console can not be undone, and so can not be allowed inside atomic blocks. The `STM` type statically enforces this restriction at compile time.

   (c) Which of these properties hold for the atomic blocks of software transactional memory? Explain why:
   - Mutual exclusion
   - Deadlock freedom
   - No starvation

   **Solution:** STM ensures mutual exclusion as well as deadlock freedom (including when multiple variables are read or written in a transaction, which is a problem for traditional lock-based approaches).

   STM suffers from starvation, as the runtime does not ensure fairness between threads: when the value in a `TVar` changes, *all* threads blocked on that `TVar` are woken up, so a thread may have continual bad luck and never be able to commit its transaction. This in particular is a problem for long-running transactions being continually aborted by shorter transactions.

2. Software transactional memory compared to locks.
   (a) What is an advantage of using atomic blocks (software transactional memory) over programming with explicit locks?

**Solution:** Examples:

- Composable atomicity. For example, in the bank accounts program using locks the `transfer` function needed to be reimplemented in a special way, rather than use the existing (and correctly working) `withdraw` and `deposit` operations.
- Composable blocking. Build operations using multiple blocking operations using `retry` and `orElse`. This is very difficult to do using locks.

(b) Give a simple example program which is easier to implement using software transactional memory compared to using explicit locks.

**Solution:** Any application which is subject to the problems of using locks, such as deadlock or lock priority inversion. The classic example is the dining philosophers.

(c) Give an example program which could be easier to implement using explicit locks compared to software transactional memory.

**Solution:** This is a question about the limitations of software transactional memory. Examples include:

- A program which relies on a fairness guarantee between threads (only applies to a locking structure with a fairness guarantee, such as `MVars`).
- An application where a thread needs to retry (or block) a transaction as well as make a visible effect at the same time. An example would be a synchronous communication channel, where both the reader and the writer must be present simultaneously for the operation to go ahead: the operation needs to block (wait for the other end to become active) as well as have a visible effect (advertise that there is a blocked thread).

3. Implementation of software transactional memory.

(a) Describe briefly how the `atomically` operation works.

**Solution:** An STM transaction works be accumulating a *log* of `readTVar` and `writeTVar` operations as they happen in the transaction, rather than storing them to main memory immediately. At the end of the transaction, the implementation *validates* the log by comparing the values read by `readTVar` during the transaction to the current contents of memory. If the values match then the `writeTVar` operations are *committed* to memory, and if not the log is discarded and the transaction runs again from the beginning.

(b) What are the performance considerations for `writeTVar`?

**Solution:** Each `writeTVar` happens in the log rather than directly to main memory. Thus it is easy to discard the effects of a failed transaction.

(c) What are the performance considerations for `readTVar`?

**Solution:** Each `readTVar` must traverse the log to see if there was an earlier `writeTVar` to the same variable. Hence `readTVar` is an $O(n)$ operation in the length of the log. Because of this, reading many `TVars` in a single transaction gives $O(n^2)$ for the whole transaction. (Computers are pretty fast at linear search, but perhaps don't read 1000 `TVars`.)

4. In Haskell the `MVar` lock implements fairness by queuing up threads blocked on the `MVar` in first-in-first-out order. A thread trying to read a value from an empty `MVar` will be queued waiting for a corresponding `putMVar`, and a `putMVar` on a full `MVar` will be queue waiting for a corresponding `takeMVar`.

   Suppose we want to implement fairness in `STM` in the same way for `TMVar`. Consider the following implementation of `TMVar`, which explicitly keeps track of the blocked `takeTMvar` and `putTMVar` operations:

```
data TMVar a = TMVar
    (TVar (Maybe a))          -- as before, the TMVar may be empty or full
    (TVar [TVar (Maybe a)])   -- list threads blocked waiting to put or take a value
```

   Will this implementation work? Consider the cases for how `putTMVar` should be implemented.

   **Solution:** This implementation will not work. There are three cases to consider:

   - The `TMVar` is empty and there are no blocked `takeMVars`. The value is stored in the `TMVar` and the operation returns.
   - The `TMVar` is empty and there are some blocked `takeMVars`. Remove the first blocked `takeMVar` from the queue and put the value into its `TVar`.
   - The `TMVar` is full. We must create a new `TVar` containing the value to be put, add this to the end of the list of blocked `putTMVars`, and wait until the `TVar` contents become empty via a corresponding `takeMVar`.

   The last case is the difficult one because we cannot have a transaction which both has a visible effect (add itself to the queue of blocked threads) *and* blocks (calls `retry`) because `retry` abandons any changes made to `TVars` in the current transaction and starts again.

5. Characteristics of software transactional memory.

   (a) When writing code using STM, you cannot perform side effects in `IO`, such as reading or writing files. Why is this restriction needed?

   **Solution:** STM relies on the ability to *undo* the effects of the atomic block when a conflict is detected, and then *retry* the transaction. Most `IO` actions, such as writing to the console, can not be undone, so the `STM` type restricts the transaction to only contain operations which can rolled-back when a transaction is retried.

   (b) How does STM guarantee the property of mutual exclusion?

   **Solution:** STM functions are executed as a single atomic transaction using the `atomically` function. To other threads, it appears that all of the modifications in the atomically block happen instantaneously.

   This is achieved by keeping track of a log of what actions a thread performs in the `atomically` section, at the end of the block, validating the log to ensure that the thread executed with a consistent view of memory, and if so, committing all changes in the log to memory.

   (c) Does STM guarantee the absence of starvation? Explain why or why not.

   **Solution:** No. A thread can be continually forced to retry, if another thread finishes its transaction and commits some change which forces the first thread to abort. This is particularly a problem if you have a long-running transaction competing with many short transactions.

(d) In some situations STM transactions can be slow. Give an example where this can occur, and explain why this happens.

**Solution:** Valid examples:

- Each `readTVar` must traverse the lock to see if it was written by an earlier `writeTVar`, which is an $O(n)$ operation.
- A transaction is woken up whenever any one of the `TVar`s in its read set changes, so calling `retry` is $O(n)$.
- Composing too many blocking operations together can cause a thread to be woken up many times. For example, if we want to wait on a list of `TMVar`s, consider
  `atomically (mapM takeTMVar ts)`; vs.
  `mapM (atomically . takeTMVar) ts`.
  In the first example the transaction is re-run from the start for every element of `ts`, so is $O(n^2)$.

6. Disgruntled with the limitations of software transactional memory, Felix makes his own version which includes the possibility to read and write files on disk. Writing to files is directly executed, and if the transaction fails, the operation is reverted by rewriting the original contents of the file back. Will this approach work? If so motivate your answer, or if not explain why or describe a problem which can be encountered.

**Solution:** This will not work because another thread can observe the intermediate state of the transaction. Thus another thread can execute with an inconsistent view of memory. Example: Thread A write a file to memory, then is descheduled. Thread B reads the contents of that file, and commits its results. Thread A resumes but detects a conflict and is forced to abort; it returns the file back to its original contents and retries. This throws away B's changes, which had been properly committed.

7. Follow the tutorial *Beautiful Concurrency.* If you have any questions, ask the TAs in the working group session.

`https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/beautiful.pdf`