

Software Transactional Memory in Haskell

APL Coursework

Carl Andersson, 0452925

s0452925@sms.ed.ac.uk

Abstract

Concurrency is becoming an ever-more important part of achieving efficiency in programs, but the standard method of handling concurrent programming, *locks*, is error-prone and plagued with difficulties. This paper looks at a transaction-based approach to handling concurrency, and in particular the software transactional memory capabilities of Concurrent Haskell. Haskell's STM abstractions allow atomic operations to be composed, and provides operators for choice and conditional blocking. The report illustrates the STM capabilities using a two-queued producer-consumer problem.

1. Introduction

Concurrency has become a corner-stone in modern applications. Whether dealing with user interfaces requiring multiple threads to achieve fluent user interaction or writing a server dealing with any number of requests simultaneously, programmers need to split their applications into smaller pieces of logic (referred to as processes or threads/light-weight processes) with the ability to run side-by-side. With the growing use of multi-processor and multi-core processor computers, concurrency has grown even more in importance if an application is to make efficient use of the resources available. Unfortunately for the programmer, creating concurrent applications is far from straight-forward in many cases. How does one ensure all threads have the same view of memory and resources external to the application, and that the various reads and writes are interleaved in such a way that the threads don't interfere with each other?

This report looks at a promising technique for ensuring thread-safety in concurrent environments using the concept of memory transactions, with focus on the software transactional memory (STM) capabilities of Concurrent Haskell. It gives details of how Haskell's STM solves the concurrency issue, looking at some of the available features, as well as instrumenting a simple example to show a transactional memory approach to a producer-consumer problem.

The report concludes that Haskell's approach to

transactional memory offers a well thought-out abstraction that greatly simplifies the use of shared variables in a multi-threaded environment, making it more flexible and modular without losing out on thread-safety. However, it also suggests that further research into the efficiency and scalability of Haskell's STM would be useful, and questions whether the approach taken here is really the right way to go.

2. Background

Over the years, a number of techniques have been created specifically for the purpose of ensuring that threads can communicate and share information in a safe manner. These techniques can be split into two general groups: *shared-memory* communication, by which threads communicate by writing to and reading from memory locations visible to the threads involved; and *message-passing* communication, where the threads share no resources but have mailboxes to which other threads can send messages containing information or commands. The former is the standard method of communication in languages like Java and C, whereas the latter is used mainly in functional programming language, having originated in Erlang[4]. The message-passing style of concurrency mechanisms is beyond the scope of this report, as the nature of the software transactional memory method of handling concurrency lies in the shared-memory domain. Should the reader be interested in finding out more about message-passing concurrency, a good overview and pointers to further information can be found at [3].

Shared-memory concurrency is based on the idea of a *lock*: if a process wishes to use some shared resource, it needs to acquire a lock for it, ensuring that it is the only process currently working with that resource. If a resource's lock is being used by some other process, the process wishing to obtain it must wait, or *block*, until the lock has been acquired before proceeding. Locks can be implemented using *mutual exclusion*, *semaphores* or *monitors*, and ensure that only one process is ever manipulating shared data, i.e. that only one process is access-

ing its *critical section*. Using locks to achieve thread-safety is a way of directly mapping which resources the developer wants to keep safe with how the application behaves. However, as applications grow so does their complexity, potentially making it difficult for developers to ascertain that thread-safety holds.

The lock-mechanisms listed above ensure that data is accessed in a safe manner, but do nothing to avoid issues of *deadlock*, which occurs when two (or more) processes holding some lock but requiring the other process' lock to proceed, leading to them both blocking forever; and *livelock*, which can occur when processes react to each others actions, e.g. when processes defer ownership of a lock to each other, never progressing. The problem is that although access to memory locations are kept safe, it is up to the programmer to ensure that the right locks are taken at the right time in the right order, while at the same time keeping the number of locks taken to a minimum to avoid hampering performance. Further to this, lock-mechanisms are not composable, meaning that although two pieces of code may be thread-safe there are no guarantees that running them both in sequence is. The non-compositionality requires the programmer to explicitly ensure that all sections of code are thread safe, not just the lower-level methods that actually perform the memory alterations, which in turn causes more overhead for the developer and increases the complexity of the code as a whole.

3. Software Transactional Memory

The previous section makes it clear that writing concurrent programs using shared-memory concurrency is far from easy, and that although the lock mechanisms allow memory to be accessed in a way that guarantees threads do not interfere with each other, it is up to the programmer to ensure that threads do not dead- or live-lock. Recent research[9] has introduced a novel way of dealing with the issue of shared-memory concurrency using *atomic transactions* in what's known as *transactional memory*. This section looks at the ideas driving transactional memory, and explores its use in Concurrent Haskell. For readers unfamiliar with Haskell, or functional programming in general, good resources can be found in [5].

3.1. Transactional Memory

The idea of using transactions to deal with sensitive operations is not new. In fact, it has long been used for database management systems to deal with the issue of concurrent data access. The driving force behind transactional memory is that of *atomicity*, i.e. that there exists some mechanism allowing a process to perform a num-

ber of operations (referred to as a *block*) with only the final result being visible by other processes, not any intermediate states. The use of atomic transactions for handling concurrent memory access abstracts the implementation of thread-safe memory operations, allowing programmers to simply specify which operations are to be performed atomically, relying on the programming language implementation to deal with the low-level mechanisms that make this possible.

One method of implementing atomic transactions it through the use of *optimistic synchronisation*, by which each process assumes that there will be no memory contention during execution, and that it therefore can perform its critical actions safely. The processes perform their actions tentatively, either retaining ordered local logs of any memory reads and writes made during the execution of an atomic block, or by making use of an update buffer. In the case of logged actions, a log is validated once the owner-process' block has finished to ensure that the process' view of memory has been consistent (i.e. that no other process has made memory accesses during the block's execution); if buffered updates are used, the process attempts to commit the changes after the block's end, failing on inconsistencies. In both cases, any inconsistencies require the results of the block to be discarded, and the whole transaction to be re-run. This is in contrast with resource locking, which is a pessimistic approach to handling concurrent memory access and assumes that there will be memory contention during execution.

The method of working with a whole block under the assumption that it will succeed, and re-running the entire block on failure, may seem like a waste of resources. Indeed, a well-written concurrency algorithm using low-level mechanisms is likely to perform better than its abstracted equivalent[1]. Transactional memory does, however, have the distinct benefit of not requiring locks to be taken at any point, making it, by design, safe from dead- and livelocks caused by lock-taking. Furthermore, the pessimism of lock-based concurrency mechanisms means that only one process can access its critical section at any one time, whereas in transaction-based mechanisms it is often possible for multiple processes to be in their critical sections simultaneously[1]. However, transactional memory by itself does not solve all the issues of concurrent programming. In particular, atomic transactions do not solve the issue of compositionality, nor do they guarantee that the operations performed within the block are appropriate for re-execution. Consider the following block of pseudo-code:

```
atomic {  
    print "Atomic access\n";  
}
```

```
    lSharedVar = 3
}
```

where `lSharedVal` is a shared memory location. If the atomic transaction fails because some other process has caused changes to happen to the memory we are wanting to modify, the entire block of code will be re-run, meaning that the `print`-statement will once again output text. While in this case the effects are trivial, one can easily imagine situations where some piece of code should not be run more than once due to its *side-effects*, i.e. irrevocable changes of state.

The issue of side-effects being re-run can be solved by disallowing such code in the atomic blocks, but it is unclear how one would go about deciding whether or not a particular operation would result in a change of state. Achieving compositionality is also less than straight-forward due to blocking not being composable. Consider the following method (inspired by a similar example in [9]):

```
Void put(Item i) {
    atomic (num_items < MAX) {
        // code to insert into some buffer
    }
}
```

A condition variable is used to ensure that the code in the atomic block is executed only if there is space in the buffer. Placing two items into this buffer and ensuring that they are neighbours is difficult, as two consecutive calls to the `put`-method does not guarantee that another process does not place an item in between the calls. Wrapping the calls in an `atomic`-block does not necessarily help unless the programmer checks that there is enough space in the buffer for both items, which breaks abstraction — synchronising on some condition requires the programmer to be aware of the internals of the atomic method.

3.2. STMs in Haskell

3.2.1. Haskell and Monads

Haskell is a pure, strictly typed functional programming language, with features such as lazy evaluation, currying and list comprehensions. In its purest form, beyond having no concept of state, side-effects are completely disallowed in Haskell: a programmer cannot modify memory locations or perform any kind of input/output. Furthermore, the purity of the language means that the programmer cannot control exactly in what order functions are called, as no execution order is enforced by the language. While these features are beneficial for the programmer,

as the former guarantees that a function will always return the same output for a given input and the latter aids execution speeds as any program can be parallelised, they are not helpful when it comes to concurrency. For shared-memory concurrency, the lack of side-effects is obviously problematic, as writing to and reading from memory is how threads communicate with each other. Similarly, even if side-effects were possible, the lack of an explicit execution order means that concurrent programs needing events to occur in a particular sequence (e.g. a money transfer requiring the source account to be debited before crediting the destination account) are difficult to write.

Fortunately, Haskell is able to circumvent these restrictions using *monads*[6], a mathematical concept taken from category theory. The details of how monads work are far beyond the scope of this report; for the purpose of this discussion it is enough to know that monads allow a programmer define operations that might be unsafe in respect to program purity. Monads are used to model state, side-effect operations etc, where the action the programmer wants to perform can be evaluated without being executed. Monads also allow function calls to be ordered explicitly, solving the sequencing issue described previously. Haskell's monads require programs and programmers to be explicit about which functions are pure and which use side-effects, keeping a code which might cause side-effects separate from pure code.

One of the main monads in Haskell is the `IO` monad, which is used to handle input and output, both of which inherently require side-effects. In order to achieve concurrency, *Concurrent Haskell* (an extension to Haskell 98, see [7]) makes use of the `IO` monad to perform a given action asynchronously. The `forkIO` function takes an `IO a`, and spawns a new thread which will perform that action, returning a thread identifier which can be used to contact the thread at some later stage. To ensure that threads spawned can work with shared data without interfering with each other, *Concurrent Haskell* uses synchronised mutable references, `MVars`. These can contain a value of some (defined) type, or be empty. *Concurrent Haskell* also provides two operations with which to modify the `MVars`: `takeMVar`, which returns the value in a given location, blocking if it is empty; and `putMVar`, which puts a given value in the given location, blocking if the location is non-empty.

3.2.2. Software Memory Transactions

Although the shared-memory concurrency mechanism in *Concurrent Haskell* works, it is a lock-based approach, and is therefore vulnerable to lock-induced dead- and live-lock. To work around this, software transactional memory has been introduced to the *Concurrent Haskell*

extension[9], giving programmers more flexibility for concurrent programming. Haskell's implementation of transactional memory makes use of the optimistic methods described in section 3.1, with no changes to memory being made unless a transaction's view of memory is consistent throughout. The library is written in C, and is built in to the Concurrent Haskell part of the Glasgow Haskell Compiler (GHC[8]). A thorough run-through of the library's implementation can be found in the original Haskell STM paper, [9].

The focus in Haskell STM lies on the ideas of *transaction variables* (TVar's), which hold some value that we wish to share between threads, and the separation of the modifications of these variables and other I/O actions. In addition to the transaction variable type, transactional memory in Haskell is exposed to the programmer through the STM monad which, similarly to the IO monad, allows programs to distinguish between pure operations and operations that may modify a memory location. STM's interface includes the following functions:

newTVar :: a -> STM (TVar a) This function creates a new STM *action*, i.e. something that may change program state, with a transactional variable of type *a*. This transactional variable can then be used to reference a mutable memory location.

readTVar :: TVar a -> STM a Given a transactional variable, this function returns the value of the memory location described by that variable.

writeTVar :: TVar a -> a -> STM a
Given a transactional variable of type *a* and a value of that same type, this function writes the value in the given location.

retry :: STM a Calling this function within an STM transaction aborts the whole transaction and begins again, without affecting any parts of memory.

orElse :: STM a -> STM a -> STM a
This function is a *choice* operation, attempting to perform the transaction in the first argument, reverting to the second on retry. If both transactions retry, the call to `orElse` retries too.

atomically :: STM a -> IO a The most important part of the interface, this takes an STM action and returns an IO action of the same type. When the IO action is performed, any memory transactions made are guaranteed to be atomic with respect to any other transactions.

From the interface, it should be clear that with the functions defined it is straightforward to create other functions that perform a given memory transaction in an atomic manner. The TVar functions makes it easy for a programmer to create and access shared memory locations; `retry` makes it possible to abort transactions completely if some condition doesn't hold; and `orElse` gives the programmer the option of specifying back-off options in case the primary option fails. Since the only way of exposing these transactions is through the `atomically` function, Haskell enforces transaction atomicity and ensures that the operations are kept free from interference. However, the interface itself does not give any indication of whether or not the implementation is safe from side-effects other than memory transactions, or if transactions can be sequenced in a safe manner. Thankfully, Haskell turns out to be the ideal setting for transactional memory. The type system has a clear distinction between pure and non-pure functions, as well as between different non-pure functions — an IO operation cannot be performed inside `atomic`, as it is not of type STM. The type system effectively takes away the whole issue of unwanted side-effects happening during transactions. Furthermore, because the transactional memory is modelled as a monad in Haskell, composition is easy: monads allow a programmer to specify a sequence of operations that are to happen within a single code block, meaning that any number of STM-operations can be composed to a single atomic transaction without the need for any external locking by the programmer.

The STM abstraction provides a well thought-out way of giving the programmer very fine-grained control over which parts of a program that need to be protected and which do not. Haskell's STM also provide valuable guarantees that unwanted side-effects cannot occur within a transaction ([13] uses the example of an IO function `launchMissiles`), meaning that even if a transaction fails, no change of state occurs. Despite all the fanciness of STM, however, it is not clear that software transactional memory is the be all and end all of concurrent programming. In particular, there are times when the transactions' *roll-back*-approach causes more overhead than it is worth, especially if there are a large amount of processes accessing the same data[2]. Transactions are an abstraction of low-level concurrency mechanisms — while the programmer does not need to worry about making memory reads and writes safe, the compiler does. The fact that transactions allow fine-grained control over which atomic operations also means that the program needs to make a greater number of low-level transactions, causing more overhead. Recent research suggests that Haskell's STM has issues with scalability, despite this being one of its major selling

points — increasing the number of processors in a machine increases the number of rollbacks needed for a transaction[2]. Furthermore, although ensuring access to shared data is kept atomic is straight-forward when using transactions, it is still a shared-variable approach to concurrency, making it more difficult to reason about and, in some cases, debug.

4. Example

To illustrate how software transactional memory works in Haskell, this section contains a small example of a concurrent program. For reasons of brevity, some aspects of the code is not discussed here; the full code can be found in appendix A. Based on the classic producer/consumer-problem, the example models a passport control at some arrivals hall, with two passport control booths servicing a never-ceasing stream of travellers. Each control booth has a separate queue, capable of containing three people at any one time, and only people in the queues will be serviced. As is standard in arrivals halls, there is no order outside of these queues — if a person finds both queues to be full, he will attempt to access them again at some later point, but there are no guarantees that some other person arriving later has not already queued up.

Queues are modelled and created as follows:

```
data Queue = MkQueue Int (TVar Int)

-- Create a queue
newQueue :: Int -> STM Queue
newQueue n =
  do { var <- newTVar 0;
      return (MkQueue n var) }
```

Queues are datatypes consisting of an integer stating its capacity and a transactional variable of type `Int` holding the current number of people in the queue. Because the `newQueue`-function returns an `STM Queue`, we can be certain that it cannot be executed outside of an atomic transaction. Once a queue has been created, we can create a booth to service that queue:

```
-- Create a passport control booth
-- to service queue q.
newBooth :: Int -> Queue -> IO ()
newBooth n q =
  do
  {
    putStrLn ("Queue " ++
              show(n) ++ " open..");
    forkIO(
      forever(
```

```
do
  {
    service n q;
    sleep 5;
  }
  )
);
return ()
}
```

```
-- Service a queue
service :: Int -> Queue -> IO ()
service n (MkQueue cap queue) =
  do
  {
    putStrLn ("Queue " ++
              show(n) ++ " servicing..");
    atomically(
      do
      {
        -- Get the current value
        lNum <- readTVar queue;

        -- Ensure queue isn't empty
        when (lNum == 0) retry;

        -- Update value
        writeTVar queue (lNum - 1);
      }
    );
    putStrLn "Serviced!"
  }
```

`newBooth` creates a booth with the (possibly non-unique) name `n` by forking a process that services the given queue forever, sleeping between one and five seconds between each call to `service`. The `service` function gets the number of people in the given queue using `readTVar`, and checks to see that there is someone to service. If the queue is empty, `retry` is called, causing the statement to restart; if there is someone in the queue, `service` services that person by decrementing the value of `queue` using `writeTVar`. The type signature of `service` states that it is to return an `IO` action, meaning that the transactions must occur in the function itself using `atomically`. The definitions of `forever` and `sleep` are given in A.

Just like it needs booths to service people, the program needs people for the booths to service.

```
-- Create a person
newPerson :: Int -> Queue -> Queue -> IO ()
newPerson n q1 q2 =
  do
```

```

{
  forkIO(
    do
      {
        putStrLn ("Person " ++
          show(n) ++ " wants to join.");
        atomically(
          (joinQ q1) `orElse` (joinQ q2)
        );
        putStrLn ("Person " ++
          show(n) ++ " joined!")
      }
    );
  return ()
}

```

```

-- Attempt to join a queue
joinQ :: Queue -> STM ()
joinQ (MkQueue cap queue) =
  do
    {
      lNum <- readTVar queue;
      when (lNum >= cap) retry;
      writeTVar queue (lNum + 1)
    }

```

Similarly to `newQueue`, `newPerson` creates a new thread symbolising a person. However, unlike the booths the people do not stay around forever — they just want to get into a queue to be serviced. `newPerson` requires there to be two (possibly non-distinct) queues for the person to join, so that if one queue is full the person can try the second one. As described in section 3.2.2, this choice is achieved using `orElse` which calls `joinQ` on the first queue, and if this retries does the same for the second queue. `joinQ` works in a way similar to `service`: the current value of the given queue counter is read using `readTVar` and compared against the queue’s capacity, retrying if the queue is deemed full. If there is space left, `writeTVar` is used to increment the queue counter. Unlike the implementation of the booths’ `service` function, `joinQ` does not perform any IO, returning instead an STM action to be performed elsewhere, which allows it to be called by `orElse`.

Finally, these functions are glued together using a `main` function:

```

main :: IO()
main =
  do
    {
      -- Create queues
      q1 <- atomically(newQueue 3);
      q2 <- atomically(newQueue 3);

```

```

-- Create booths
newBooth 1 q1;
newBooth 2 q2;

-- Create people forever.
loopIncr
  (\x ->
    do
      {
        newPerson x q1 q2;
        sleep 3
      }
    ) 0;
  return ()
}

```

Where `loopIncr` is a function that calls a given function forever, passing it an ever-increasing integer value. As is to be expected, `main` creates two queues, each serviced by a separate booth, before creating people randomly every one to three seconds.

Figure 4 shows screenshots of part of the code, the compilation and the passport queue example in action. 1(c) shows the program during execution, with both booths serving queues and a number of people waiting to get in the queue. It is obvious from the figure that the program follows the high-level description given, with people joining the queues as space becomes available, out of order, with whichever person thread attempts to access the queue variable first when it is decremented by a booth thread getting to queue up and exit.

5. Resources

This section lists a number of resource regarding transactional memory in Haskell.

Composable Memory Transactions[9] was the paper the first introduced the implementation of STM in Haskell, and provides a good overview of the background of transactional memory, as well as a relatively detailed description of Haskell’s STM semantics and implementation. The paper also looks at how the lock-based MVars can be re-expressed using STM, and how these can be extended to provide more advanced communication methods.

Beautiful Concurrency[13] is taken from the book *Beautiful Code*, and is a less formal look at Haskell’s STM, describing its features while at the same time introducing Haskell as a language. The chapter gives examples of some issues found when using lock-based concurrency mechanisms,

```

emacs@daxter.inf.ed.ac.uk
File Edit Options Buffers Tools Haskell Help

return ()
}

-- Create a passport control booth to service queue q.
newBooth :: Int -> Queue -> IO ()
newBooth n q =
do
  {
    putStrLn ("Queue " ++ show(n) ++ " open..");
    forkIO(
      forever(
        do
          {
            service n q;
            sleep 5;
          }
      );
    return ()
  }

-- Attempt to join a queue
joinQ :: Queue -> STM ()
joinQ (MkQueue cap queue) =
do
  {
    lNum <- readTVar queue;
    when (lNum >= cap) retry;
    writeTVar queue (lNum + 1)
  }

-- Service a queue
service :: Int -> Queue -> IO ()
service n (MkQueue cap queue) =
do
  {
    lNum <- readTVar queue;
    when (lNum >= cap) retry;
    writeTVar queue (lNum + 1)
  }

Main.hs (Haskell Doc Ind)--L65--37%

```

(a) The code.

```

Terminal - s0452925 @ daxter:/home/s0452925/workspace/apl/src
File Edit View Terminal Go Help
[daxter]s0452925: ghc Main.hs -package stm -o apl
[daxter]s0452925:

```

(b) The undramatic code compilation.

```

Terminal - s0452925 @ daxter:/home/s0452925/workspace/apl/src
File Edit View Terminal Go Help
Person 33 joined!
Queue 1 servicing..
Serviced!
Person 24 joined!
Queue 2 servicing..
Serviced!
Person 22 joined!
Person 34 wants to join.
Queue 2 servicing..
Serviced!
Person 34 joined!
Queue 1 servicing..
Serviced!
Person 29 joined!
Person 35 wants to join.
Queue 2 servicing..
Person 35 joined!
Serviced!
Person 36 wants to join.
Person 37 wants to join.
Person 38 wants to join.
Person 39 wants to join.
Queue 2 servicing..
Serviced!

```

(c) The program in action.

Figure 1: The arrival passport queue example.

and concludes with an STM-based solution to the *Santa Claus Problem*[18] in Haskell.

Lock Free Data Structures using STM in Haskell

compares the implementation of Haskell versions of Java's `ArrayBlockingQueue`, one using a lock-based approach, the other STM. Although quite heavy on code-use, the paper is an interesting comparison of how the code for the two versions differ, and what the strengths of STM is.

Transactional Memory and Concurrency[14, 15] is a presentation by one of the implementers of Haskell's STM, describing the background of the problem STM wants to solve, why locks are an issue, and what the features promises of STM are. The presentation is not specific to Haskell, but rather a general introduction to transactional memory.

6. Related Work

As discussed in section 2, there are two main methods of dealing with concurrency: shared-memory and message-passing. With the former, lock-based mechanisms for ensuring accesses to shared data are achieved synchronously has been the standard method for over 30 years[14]. The latter is more modern, stemming from the extreme concurrency and parallelism needed in telecoms applications whereby no threads or processes share any data at all.

Software transactional memory has been implemented for a number of languages, including *C*[16], *C++*[10], *C#*[12] and *OCaml*[11]¹. However, the majority of STM implementations are external libraries rather than extensions to the language. The main difference between these and the implementation discussed here is the issue of compositionality: although they all perform concurrent memory access using transactions, the transactions cannot be composed into larger ones². Further to this, no other implementation includes a choice operator, and not all include a retry mechanism for cancelling transactions on some condition.

7. Conclusion

This report has focused on the implementation of software transactional memory in the functional programming language Haskell. Transactional memory allows programmers to write applications that atomically modify data in a lock-free manner, removing the need for

explicitly acquiring access to a resource in order to be able to work with it. This in turn removes the danger of taking too many, too few, or the wrong locks, making lock-based deadlock a thing of the past. Haskell's implementation of STM goes one step further by adding compositionality and banishing other side-effects (e.g. IO) from within a transaction. Haskell STM makes it trivial to write applications that are not only thread-safe, but modular — larger transactions can be built out of smaller ones without having to expose to the programmer any of the internals of the functions. The use of a choice operator and the ability to cancel transactions based on some condition gives the programmer a great deal of flexibility in terms of how to access shared data. Furthermore, Haskell's type system's strictness when it comes to separating side-effect functions from pure ones means that transactions cannot be performed inside pure code, thereby discouraging the performing of side-effect operations unless it is actually necessary. However, it should be noted that the use of greater abstractions does not necessarily lead to better code or the complete elimination of bugs.

Haskell's STM makes use of optimistic transactions, which make the implementation more straight-forward and allows it to be completely lock-free. Unfortunately, judging by the results in [2], it appears that this comes at the cost of efficiency in some cases, with roll-backs happening to often or too slowly. However, other research suggests that with certain compiler optimisations Haskell STM can be made more efficient, allowing the fine-grained atomicity while sacrificing only a small amount in terms of efficiency[15]. Furthermore, these optimisations allow for greater scalability, giving Haskell STM a possible edge over lock-based concurrency mechanisms. These conflicting stories suggest that it would be beneficial for the Haskell and concurrency community to look further into the efficiency and scalability of the STM solution.

Finally, while it is probable that software transactions will become more common in the future, it is unclear whether the transaction-based concurrency mechanisms are the right direction to go in. As scalability becomes more and more important, it seems as though concurrent programming should be looking at sharing less and less data, not just working around thread-contention. While the use of transactions is definitely an improvement over lock-based methods in terms of ease of implementation for the programmer, Erlang-style "share-nothing" concurrency feels like a much better approach to take, as it minimises the possibility of threads contending in the first place.

¹A more complete list can be found on Wikipedia's page on STM: http://en.wikipedia.org/wiki/Software_transactional_memory

²The one exception from this is *Concurrent ML*[17] (CML), in which *actions* can be composed in a limited fashion.

8. References

- [1] Ayguade, E; Cristal, A; Gagliardi, F; Smith, B; Harris, T; Unsal, O. S; Valero, M, *Transactional Memory: An Overview*, IEEE Micro, vol. 27, no. 3, pp. 8-29, May/Jun, 2007
- [2] Cristal, A; Harris, T; Perfumo, C; Sonmez, N; Unsal, O. S; Valero, M, *Dissecting Transactional Executions in Haskell*, TRANSACT 2007
- [3] Akhmechet, S, *Erlang Style Concurrency*, <http://www.defmacro.org/ramblings/concurrency.html> (retrieved 05/03/08)
- [4] Open-source Erlang, *Erlang*, <http://www.erlang.org> (retrieved 13/03/08)
- [5] Fasel, J; Hudak, P; Peterson, J, *A Gentle Introduction to Haskell* <http://www.haskell.org/tutorial/> (retrieved 08/03/08)
- [6] Fasel, J; Hudak, P; Peterson, J, *A Gentle Introduction to Haskell: About Monads* <http://www.haskell.org/tutorial/monads.html> (retrieved 08/03/08)
- [7] Finne, S; Gordon, A; Peyton Jones, S, *Concurrent Haskell*, 3rd ACM Symposium on Principles of Programming Languages, pp295-308, Jan 1996
- [8] GHC, *The Glasgow Haskell Compiler*, <http://www.haskell.org/ghc/> (retrieved 12/03/08)
- [9] Harris, T; Herlihy, M; Marlow, S; Peyton Jones, S, *Composable Memory Transactions*, 10th ACM Symposium on Principles and Practice of Parellel Programming, pp48-60, June 2005
- [10] Intel, *C++ STM Compiler, Prototype Edition*, <http://softwarecommunity.intel.com/articles/eng/1460.htm> (retrieved 13/03/08)
- [11] Li, Z, *coThreads*, <http://cothreads.sourceforge.net/> (retrieved 13/03/08)
- [12] Microsoft Research, *C# software transactional memory*, <http://research.microsoft.com/research/~downloads/default.aspx> (retrieved 13/03/08)
- [13] Peyton Jones, S, *Beautiful Concurrency*, extract from *Beautiful Code*, O'Reilly 2007
- [14] Peyton Jones, S, *Transactional Memory and Concurrency*, presentation, O'Reilly Open Source Convention, July 2007, <http://www.blip.tv/file/317758/> (retrieved 11/03/08)
- [15] Peyton Jones, S, *Transactional Memory and Concurrency*, slides, O'Reilly Open Source Convention, July 2007, <http://research.microsoft.com/~simonpj/papers/stm/STM-OSCON.pdf> (retrieved 11/03/08)
- [16] Protti, D, *LibCMT*, <http://libcmt.sourceforge.net/> (retrieved 13/03/08)
- [17] Reppy, J. R, *Concurrent ML: Design, application and semantics*, LNCS 693, Springer-Verlag, Berlin, 1993, pp165-198
- [18] Trono J. A, *A new exercise in concurrency*, SIGCSE Bulletin, vol 26 no. 3, pp810, 1994

A. Sample Code

```
{-
  STM version of a producer-consumer problem using two queues.

  The general layout of the problem is this:
  * A passport control at an airport has two queues
  * The queues have a fixed capacity of 3 people.
  * People attempt to join the queues at random times between 1-2 seconds.
  * The service time at the control booths is between 1-5 seconds.
-}
module Main where

import Control.Concurrent
import Control.Concurrent.STM
import Control.Monad
import IO
import Random

data Queue = MkQueue Int (TVar Int)

-- Create a queue
newQueue :: Int -> STM Queue
newQueue n = do { var <- newTVar 0; return (MkQueue n var) }

-- Create a person
newPerson :: Int -> Queue -> Queue -> IO ()
newPerson n q1 q2 =
  do
  {
    forkIO(
      do
      {
        putStrLn ("Person " ++ show(n) ++ " wants to join.");
        atomically( (joinQ q1) `orElse` (joinQ q2) );
        putStrLn ("Person " ++ show(n) ++ " joined!")
      }
    );

    return ()
  }

-- Create a passport control booth to service queue q.
newBooth :: Int -> Queue -> IO ()
newBooth n q =
  do
  {
    putStrLn ("Queue " ++ show(n) ++ " open..");
    forkIO(
      forever(
        do
        {
          service n q;
          sleep 5;
        }
      )
    )
  }
```

```

        }
    )
);
return ()
}

-- Attempt to join a queue
joinQ :: Queue -> STM ()
joinQ (MkQueue cap queue) =
do
{
    lNum <- readTVar queue;
    when (lNum >= cap) retry;
    writeTVar queue (lNum + 1)
}

-- Service a queue
service :: Int -> Queue -> IO ()
service n (MkQueue cap queue) =
do
{
    putStrLn ("Queue " ++ show(n) ++ " servicing..");
    atomically(
        do
        {
            -- Get the cap and the queue
            --(MkQueue cap queue) <- q;

            -- Get the current value
            lNum <- readTVar queue;

            -- Ensure queue isn't empty
            when (lNum == 0) retry;

            -- Update value
            writeTVar queue (lNum - 1);
        }
    );
    putStrLn "Serviced!"
}

main :: IO()
main =
do
{
    -- Create queues
    q1 <- atomically(newQueue 3);
    q2 <- atomically(newQueue 3);

    -- Create booths
    newBooth 1 q1;
    newBooth 2 q2;
}

```

```

-- Create people forever.
loopIncr
  (\x ->
    do
      {
        newPerson x q1 q2;
        sleep 3
      }
    ) 0;
return ()
}

{--
Utility functions
--}
-- Loop forever.
forever :: IO () -> IO ()
forever a = a >> forever a

-- Loop on a function, passing it an ever-
-- increasing integer.
loopIncr :: (Int -> IO ()) -> Int -> IO ()
loopIncr a n =
  let np = n + 1 in do {a np ; loopIncr a np}

-- Sleep between 1 and n seconds.
sleep :: Int -> IO ()
sleep n = do {lTime <-randomRIO (1, (n * 1000000) :: Int); threadDelay lTime}

```