Software Transactional Memory in Haskell

Submission for the Informatics Research Review module,
University of Edinburgh, 19 January 2017

Lukas Convent

Abstract

To deal with data races in a multi-threaded setting, using locks has many disadvantages. Most noticeable, they are hard to get right and programs relying on them are often not composable. As a remedy, software transactional memory uses re-executable transactions as a way of ensuring correctness and composability. Memory operations and multi-threading is already part of Haskell through the use of monads. This review focuses on how monads and in general Haskell's type system is used to get strong static guarantees about the correct usage of STM, and therefore about programs being free of data races and deadlocks.

1 Introduction

The standard observation (yet) holds: Concurrent programming is notoriously difficult. Sharing resources between multiple threads requires thorough coordination, for which different models and tools have been developed. First introduced by Shavit et al. [14], software transactional memory (STM) is an approach based on atomic transactions similar to the ones used in the database field. Optimistically, all transactions are executed concurrently but are eventually only committed if the executing threads's memory view is consistent with the actual memory at commit-time. Harris et al. have enriched the usual transactional memory operations by blocking and choice mechanisms, resulting in a composable transactional memory model [7]. Furthermore, they embedded this model in Haskell and implemented it as an extension to the Glasgow Haskell Compiler.

The focus of this review lies on this embedding of STM in a pure functional language like Haskell and how this enables strong static guarantees by using Haskell's type system, in particular through monads. We begin by introducing STM following Harris et al. [7], thereby demonstrating its advantages and the challenges of implementing it. Then we focus on the embedding of STM into Haskell and point out at which aspects the implementation excels, thereby contrasting the Haskell implementation to ones in other languages. We conclude by evaluating further directions taken from the presented STM model.

Shared-memory concurrency. We briefly introduce some terminology for shared-memory concurrency. Like the concept of locks, the STM model renders support for memory-sharing between multiple threads. A **thread** is a sub-program whose execution depends on a scheduler that manages the distribution of system resources. One assumes nondeterministic scheduling which means that the effective execution of multiple threads can be modeled as a **nondeterministic interleaving** between the individual threads' executions. This nondeterminism can produce different orders of accessing the same part of memory, leading to different results depending on the scheduling. Such a situation is called a **data race**. In the following, we give a brief overview of the concept of traditional locking and present the difficulties it brings along. We then focus on the STM model and how it can help to overcome these difficulties.

2 Locks and their challenges

Traditional locking allows identifying critical regions which introduce data races when interleaved. Each group of critical regions is assigned a single lock that needs to be acquired first by a thread before entering one of these critical regions. If the lock is returned, the scheduler picks one of the threads of the **waiting pool** and gives it the lock. This way, some nondeterminism in the interleaving is cut out because critical regions cannot be interleaved any more.

Sometimes a thread, being in the middle of a critical region, must wait for a certain condition to be fulfilled. After returning its lock, thereby giving other threads the chance to advance and fulfill the condition, the thread is placed in the **blocked pool** assigned to the lock. They can be awoken (unblocked) and therefore added back to the waiting pool by other threads which see the chance of the condition now to be fulfilled. This is called a **wake-up call**.

Manually identifying critical regions and assigning locks to them is error-prone, leading to the following problems as identified by Peyton Jones [9] in a list as follows:

- **Too few locks**: This results in data races.
- **Too many locks**: This results in the prevention of concurrency due to too little interleaving or even deadlocks.
- **Taking locks in the wrong order**: When threads A and B want to acquire both locks L1 and L2, but A takes first L1 and B first L2, with both waiting for the respective second lock, they are dead-locked.
- **Error recovery**: If a thread crashes in the middle of a critical region, it must leave the critical region in a consistent state and return the lock afterwards.
- **Lost wake-up calls**: If a thread is waiting for a condition to be fulfilled, it needs to be woken up by a wake-up call to eventually make progress again.

Besides these challenges for the programmer, there is a fundamental problem with programs whose concurrency is controlled by locks: They are not composable. Although program fragments are correct for themselves, reusing them without breaking the abstraction principle is not possible in general. The reason is that the locking mechanism should not be visible to the user. For some useful extensions though, access to these internal locks is required. Take for example a counter with an increment operation which shall be performed atomically:

```
class Counter {
   int counter;

   public void increment() {
      synchronized (this.internalLock) {
         counter ++;
      }
   }
}
```

The `increment` operation on an object is protected by an internal lock of that object. However, an (external) implementation of an operation `incrementBoth` cannot be implemented atomically:

```
void incrementBoth(Counter a, Counter b) {
   a.increment(); b.increment();
}
```

To achieve atomicity, we need to acquire *both* internal locks before we perform any increment. Since these locks are internal to the counter objects, this would break the abstraction principle.

3 Software Transactional Memory

The main difference to traditional locking is the mechanism of preventing data races. Using locks, of all critical regions protected by a lock, only one can be executed at any point of time, thereby avoiding interference. The STM model acts as a buffer between memory and memory-accessing threads, allowing the concurrent execution of multiple critical regions. These critical regions are called transactions and are nested within code as follows (here marked by `atomic`):

```
block1;
atomic {
   block2;
}
block3;
```

When a thread reaches such an `atomic` block, the transaction is tried to be executed locally. This is where the buffer functionality comes in: All memory writes are not performed directly to the memory but to the local buffer, called the **log** of the transaction. Furthermore, memory reads are logged, thereby establishing a local memory view. When the local execution of the transaction finishes, the log is validated against the actual memory. If the transaction's memory view is consistent with the actual memory, all writes are transcribed instantaneously to the memory, the transaction is said to be **committed**. In the case of inconsistency, the log is cleared and the transaction re-executed.

Because of the possibility of re-execution, STM constrains the type of code to be executed within a transaction. Apart from memory operations, no side-effects are allowed, because they cannot be undone.

As with critical regions, a transaction might need to wait until a certain condition is fulfilled. STM allows the transaction to be re-executed on demand by a so-called `retry` command (e.g. when a required condition is found not to be fulfilled). The transaction though is not directly re-executed. Only if one of the accessed variables changes in memory, there is the chance of the condition being fulfilled. When such checks are made at the beginning of the transaction, this mechanism guarantees efficient and automatic wake-up calls.

It is also possible to specify multiple alternative transactions and let only one of them commit. The primitive introduced by Harris et al. [7] is called `orElse` and composes two transactions together (using it as a binary operator). If the left transaction succeeds, the right transaction is ignored. Only if the left transaction blocks (triggered by `retry`), the right transaction is executed.

4 Embedding in Haskell

We discuss in this section the embedding of STM in Haskell as it is described and implemented by Harris et al. [7]. After giving a short overview of monads in Haskell, we introduce the `STM` monad as a transaction-level version of the `IO` monad.

Monads. Haskell is a pure functional programming language, i.e. functions have no side-effects. However, the concept of monads, more specifically the `IO` monad, makes it possible to describe interaction with the external world. In the following, we briefly describe the general concept of monads. A type constructor of kind[1] `* -> *`, i.e. a parametric type like `[]` (lists) or `Maybe`, is said to have a **monadic structure** (and is briefly called a **monad**) if it allows the sequencing of computation steps. How this sequencing happens is part of the monadic structure and we give two examples for the `[]` and `Maybe` monads.

In the case of `[]`, a computation step from an input `x :: [a]` to an output of type `[b]` is specified through a function `f :: a -> [b]` and is carried out by `concat (map f x)`. In the case of `Maybe`, a computation step from an input `x :: Maybe a` to an output of type `Maybe b` is specified through a function `f :: a -> Maybe b` and is carried out as follows: If `x` is `Nothing`, the output is `Nothing`, too. Otherwise, the output is `Just (f x)`.

These two examples demonstrate how computation chains can be established by transforming values *within* the monad container. Next to this binding mechanism, a monadic structure must also provide the possibility of lifting a value into the monad container. In the case of `[]`, a value `x :: a` is lifted into the monad container as `[x] :: [a]`. In the case of `Maybe`, a value `x :: a` is lifted into the monad container as `Just x :: Maybe a`.

In general, we can identify a monadic structure of a type by instantiating the `Monad` type class, where `return` is the lifting operation and `>>=` is the bind operation.

```
class Monad where
   return :: a -> m a
   (>>=)  :: m a -> (a -> m a) -> m b
```

IO monad. The `IO` monad allows the *description* of imperative programs. Instances of the `IO` monad can be seen as actions resulting in a (possibly nondeterministic) value when executed, e.g. an action of type `IO Int` will result in an integer value. These actions are never directly executed by Haskell code, because this way nondeterministic results would “leak” into the code, thereby breaking the purity of Haskell. Instead, a (possibly large, composed) action will be delivered which is then executed at a lower level at runtime (with access to input/output resources). While `return` allows putting a value into any monad container, in the concrete case of `IO` there is no way of taking a value out of the container (i.e., executing the action). `IO` is an abstract data type which provides means of user interaction, memory operations as well as operations for introducing concurrency:

```
putStr   :: String -> IO ()        -- print string

newMVar  :: a -> IO (MVar a)       -- allocate cell
takeMVar :: MVar a -> IO a         -- read from cell
putMVar  :: MVar a -> a -> IO ()   -- write to cell

forkIO   :: IO a -> IO ThreadId    -- spawn thread
```

The `MVar` cells behave like bins with space for a single value. When a thread wants to read from a cell, it thereby takes out the value. If a thread wants to read from an empty cell or write to a full cell, it blocks. This `MVar` primitive enables lock-based concurrency control, as a single cell can already be interpreted as a lock. We conclude the presentation of the `IO` monad by a simple example of concurrency control through a lock. For a given lock (a bin that contains a unit token `()` or not), the critical region can only be executed in a mutually exclusive way. In the `main` procedure, the main

[1]The kind of a type constructor is its “type”, where `*` stands for the kind of all instantiable types

thread and a newly forked thread concurrently execute the critical region, excluding each other via a lock. The `do` notation is syntactic sugar in Haskell for writing monadic programs in an imperative style, as demonstrated on the example of the `main` procedure.

```
criticalRegion :: MVar () -> IO ()
criticalRegion lock = do
  token <- takeMVar lock            -- take lock
  putStr "critical␣region..."       -- critical block
  putMVar lock token                -- return lock


main :: IO ()                       main :: IO ()
main = do                           main =
  lock <- newMVar ()                  newMVar () >>=
  forkIO (criticalRegion lock)        \lock -> forkIO (criticalRegion lock) >>=
  criticalRegion lock                 \_ -> criticalRegion lock
```

STM monad. Having introduced a way of implementing lock-based concurrency in Haskell, we now take a look at how STM is established on top of this infrastructure. While the `IO` monad allows the description of computation that includes side-effects, the `STM` monad allows the description of computation that includes at most memory-effects. Such a computation is a transaction which can be re-executed if necessary. Similar to `IO`, the `STM` monad acts as an abstract interface which allows concatenation (monadic structure) and provides memory operations.

```
newTVar   :: a -> STM (TVar a)        -- allocate cell
readTVar  :: TVar a -> STM a          -- read from cell
writeTVar :: TVar a -> a -> STM ()    -- write to cell
```

These memory operations using `TVar` cells work like standard memory cells (in contrast to the `MVar` cells acting as bins).

Such a composed transaction can now be transformed into an `IO` action by using the following procedure:

```
atomically :: STM a -> IO a
```

On execution, this `IO` action attempts to perform the transaction. Here another major advantage of using the Haskell type system and its implied abstraction becomes visible: Not only are transactions guaranteed to perform *at most* memory operations, but also memory transactions can *only* occur within transactions. In standard programming languages like Java, memory operations are not as easily isolated because they are at the core of imperative programs. STM is therefore a good example of how in Haskell certain levels of programming (in this case: pure vs. memory-effective vs. general-IO) can be separated and transformed into each other in a precisely defined manner.

Blocking and providing alternatives. When coming to a point at which the transaction needs to be re-executed, one can abort the transaction by `retry :: STM a`. We introduce this primitive transaction together with `orElse :: STM a -> STM a -> STM a` which composes alternative transaction together. As described, a composed transaction `t1 ‘orElse‘ t2` will first try to execute `t1` and only if `t1` fails try to execute `t2`.

Exceptions. What happens if an exception is thrown inside of a transaction? There are two plausible possibilities: A transaction can be seen as completed or as failed. In the first case, a commit would be attempted. In the second case, an abortion semantics is used, i.e. the transaction is considered as failed. For their implementation of Haskell STM, Harris et al. chose the abortion semantics, arguing that exceptions in Haskell are usually considered as errors and not considered for controlling the program flow [7]. However the design choice is made, the exception propagates outside of the transaction and can be caught and handled as it is expected.

Operational semantics. Although Haskell itself does not have a formal semantics [8], Harris et al. give an operational semantics to specify their STM model for Haskell [7]. It defines precisely how a transaction is executed and how its result effects the main control flow of a concurrent program.

5 Improvements and Further Work

There have been some improvements and suggestions on how to improve the STM model, sometimes with respect to Haskell. We consider some developments and focus specifically on their meaning for an embedding in Haskell.

Harris and Stipić have proposed a refinement of the concept of transaction through the use of abstract nested transactions (ANT) [6]. The key idea is that the programmer can identify pieces of code within transaction which are likely to collide and annotate them as ANTs. When validating a transaction and some collision is detected, instead of re-executing the whole transaction it suffices to re-execute an ANT. Although introducing some overhead for additional logs, this can lead to faster code because of fewer re-executions of whole transactions. Integration into Haskell should be quite straightforward, because essentially only a new kind of primitive `ant :: STM a -> STM a` needs to be introduced which triggers execution behavior as described in [6].

Liyang HU et al. [12] have marked a subset of concurrent Haskell with provided semantics. They give an implementation of a compiler that translates the concurrent code to executable low-level code which contains no more notion of transactions. One of their major contributions is the correctness proof of this compiler which guarantees the adherence to the specified semantics.

Borgström et al. formulate a calculus for STM in Haskell [1], basing it on the ambient calculus [2]. By providing a core fragment of concurrent Haskell with a type system and operational semantics, they establish a framework for formal reasoning about both programs with and without side-effects. Because the type system separates pure and effectful computation, reasoning about such code can be separated as well, allowing usual high-level algebraic equivalences for reasoning about pure computation and rules which include the heap for effectful computation.

Implementations. An implementation of STM Haskell as described by Harris et al. was given by the authors as a complement to their publication [7]. Next to a top layer in Haskell, the implementation is based on a library in C to keep track of logs and to commit. Sabel uses the process calculus CSHF as a model for his implementation, first showing that it is applicable and then carrying out the concrete implementation of STM in Haskell [13]. Recent work by Le et al. consists of an implementation which is based on ideas of TL2 [3], which the authors find simpler and more efficient than the original implementation [10]. Another very recent comparison of different approaches was given by Ghosh et al., who took a closer view at how the scheduling order effects the overall performance [5].

6 Conclusion

In which way concurrency should be realized is very much debated, but the most prominent paradigms are multi-threading and message-passing. While there exist arguments for abandoning multi-threading in general – mainly because the introduced nondeterminism is very hard to manage [11] – multi-threading can also be seen as the natural generalization of single-threading. Instead of giving up this paradigm in general, STM tries to eliminate some of the disadvantages of the traditional method to tame the introduced nondeterminism, namely locks.

Because setting up the right locks is so hard, the focus of the STM method lies on correctness rather than efficiency. By identifying compact transactions and checking for invalid states as early as possible within a transaction, STM gives full guarantee of deadlock-freedom and freedom of data races at the same time. The main drawback of programs relying on locks, not being composable, is taken away by STM as well. By centralizing memory access, there is a single interface (through validation and consequent commit) which guarantees consistent program states. The local nature of locks therefore disappears and enables the programmer to extend concurrent programs without adjusting their locking mechanism. STM introduces some overhead for writing the logs and re-executions are sometimes performed, however, benchmarks as carried out by Discolo et al. show that the "STM implementation consistently outperformed the locking version when two or more processors were available" [4]. Although a general judgment is of course not possible, there are indications that the overhead of STM for many applications does not effect the efficiency too heavily.

Haskell is a very well-suited host language for implementing STM. Monads are a natural description mechanism for imperative programs which is not defined as a language primitive but in terms of the language itself. Because monads are descriptive, their instances (descriptions) can be interpreted, transformed (`atomically :: STM a -> IO a`) and executed (at runtime) in various ways. Monads allow the restriction to certain programming elements, such as pure computation with at most memory-effects (`STM` monad). This separation between different levels of code serves also as an aide for reasoning about concurrent programs, as it is guaranteed by the type system that certain properties inherent to a certain class programs hold.

References

[1] Johannes Borgström, Karthikeyan Bhargavan, and Andrew D. Gordon. "A compositional theory for STM Haskell". In: *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell.* 2009, pp. 69–80.

[2] Luca Cardelli and Andrew D. Gordon. "Mobile ambients". In: *International Conference on Foundations of Software Science and Computation Structure.* 1998, pp. 140–155.

[3] Dave Dice, Ori Shalev, and Nir Shavit. "Transactional locking II". In: *International Symposium on Distributed Computing.* 2006, pp. 194–208.

[4] Anthony Discolo et al. "Lock free data structures using STM in Haskell". In: *International Symposium on Functional and Logic Programming.* 2006, pp. 65–80.

[5] Ammlan Ghosh and Rituparna Chaki. "Implementing Software Transactional Memory Using STM Haskell". In: *Advanced Computing and Systems for Security.* 2016, pp. 235–248.

[6] Tim Harris et al. *Abstract nested transactions.* Microsoft Research. 2007.

[7] Tim Harris et al. "Composable memory transactions". In: *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming.* 2005, pp. 48–60.

[8] Paul Hudak et al. *A History of Haskell: Being Lazy With Class*. 2007.

[9] Simon Peyton Jones. “Beautiful concurrency”. In: *Beautiful Code: Leading Programmers Explain How They Think* (2007), pp. 385–406.

[10] Matthew Le, Ryan Yates, and Matthew Fluet. “Revisiting software transactional memory in Haskell”. In: *Proceedings of the 9th International Symposium on Haskell*. 2016, pp. 105–113.

[11] Edward A. Lee. “The Problem with Threads”. In: *IEEE Computer* 39.5 (2006), pp. 33–42.

[12] H. U. Liyang and Graham Hutton. *Towards a Verified Implementation of Software Transactional Memory*. 2009.

[13] David Sabel. “A Haskell-Implementation of STM Haskell with Early Conflict Detection.” In: *Software Engineering (Workshops)*. 2014, pp. 171–190.

[14] Nir Shavit and Dan Touitou. “Software transactional memory”. In: *Distributed Computing* 10.2 (1997), pp. 99–116.