

CHRISTIAN-ALBRECHTS-UNIVERSITY

MASTER THESIS

---

# Alternative Software Transaction Implementation in Haskell

---

*Author:*  
Lasse Folger

*Supervisor:*  
Dr. Frank Huch

*A thesis submitted in fulfillment of the requirements  
for the degree of Master of Science  
in the*

Programming Languages and Compiler Construction  
Department of Computer Science

January 5, 2017



## Declaration of Authorship

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel, January 5, 2017

---



Christian-Albrechts-University

## Abstract

Faculty of Engineering  
Department of Computer Science

Master of Science

## Alternative Software Transaction Implementation in Haskell

by Lasse Folger

TODOTODOTODOTODOTODOTODOTODO

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too...



## *Acknowledgements*

TODOTODOTODOTODOTODOTODO

The acknowledgments and the people to thank go here, don't forget to include your project advisor. . .





# Contents

<b>Declaration of Authorship</b>	<b>iii</b>
<b>Abstract</b>	<b>v</b>
<b>Acknowledgements</b>	<b>vii</b>
<b>1 Motivation</b>	<b>1</b>
<b>2 Introduction</b>	<b>3</b>
2.1 Software Transactional Memory . . . . .	3
2.2 Implementation . . . . .	5
2.2.1 Computation Phase . . . . .	5
<b>A Frequently Asked Questions</b>	<b>7</b>
A.1 How do I change the colors of links? . . . . .	7
<b>Bibliography</b>	<b>9</b>



# List of Figures



# List of Tables



# List of Abbreviations

<b>LAH</b>	List Abbreviations Here
<b>WSF</b>	What (it) Stands For
<b>STM</b>	Software Transactional Memory
<b>ACID</b>	Atomicity Consistency Isolation Durability





## Chapter 1

# Motivation

Modern computer architecture includes multicore processors. To utilize these multi-core system to their full extend, concurrent and parallel programming is needed. By this new challenges arise. One challenge is the logical issue of splitting the problem in smaller problems which can be processed by different threads in parallel. Additionally there are technical challenges. A new scheduler is needed and hardware accesses (Printer, Display, etc.) need to be sequential for example. These are challenges the operating system usually handles. There are other challenges the operating system cannot handle, because they are specific for every program.

The most discussed challenge is the synchronization. If a program works with multiple threads, these threads usually communicate. Communications means to exchange data. Even a simple statement like an assignment can cause problem when used in the parallel threads. The problem is that these operations are non atomic operations. Thus  $(x = x + 1)$  consist of three parts. first reading the old value, second adding 1, and third write the new value. This means two threads in parallel can both read the old value, then both add 1 to the old value, and then write the new value. The new value is the initial value incremented by 1, even though two threads executed an increment operation on this value. This non intended behaviour is called *lost update*. The efforts to avoid this non intended behaviour are called synchronization.

Even though multicore processors are new, the research in the field of synchronization has a long history, starting with (Dijkstra, 1965), which introduces the most basic synchronization tool, the semaphore. The semaphore is an abstract datatype which holds an Integer and provides two *atomic* operations, P and V. If the value of the semaphore is greater than 0, P decrements the semaphore. If the value of the semaphore is 0 the thread that evoked P is suspended. When a thread evokes V the value of the semaphore is increased and in the case another thread is currently suspended, because it called P on the semaphore, that thread is awakened. After the thread is awakened, it tries P again.

This seem to be a simple construct, but its capabilities are enormous. It is highly complex to use a semaphore correctly. The main problem of semaphores is the so called deadlock<sup>1</sup>. This means there is a schedule, where no progress of the system is possible, because all threads are waiting for a semaphore. The term deadlock is not exclusive for semaphores. It is used for all blocking mechanisms. To avoid such deadlocks is very hard even when using one or few semaphores. It is nearly impossible to avoid deadlocks when you try to compose semaphore based functions.

To avoid the problems of semaphores while maintaining the expressiveness of semaphores the so called software transactions were introduced (Harris et al., 2005). Software transactions are inspired by the long known database transactions (Gray

---

<sup>1</sup>In the course of this thesis I will refer to deadlocks as a static property rather than a state of a system.

and Reuter, 1992). Software transactions provide an interface to program with single element buffers. If you are using this interface the underlying implementation ensures the so called *ACI(D)* properties. **A** for atomicity. This means a transactions appears to be processed instantaneous. **C** for consistency. This means that a consistent view of the system is always guaranteed. **I** stands for isolation. This means the programmer does not need to worry about concurrency and every thread can act as if it were the only thread. **D** stand for durability, but is relevant only for data base transactions.

There is a stable implementation for software transactions in Haskell, namely Software Transactional Memory (called STM in the following). STM provides the *ACI(D)* properties by optimistically executing the transaction. If a conflict is detected, the changes are discarded and the transactions is restarted (a so called rollback). This works, but is not optimal with regards to efficiency and performance. There are two problems. First the conflict detection. Sometimes the implementation detects a conflict and evokes a rollback, even though it is not necessary. The second problem is the rollback mechanism. Regardless of the conflict, always the whole transaction is reexecuted. This includes operations on data that has not changed, thus an unnecessary recomputation. These problems are discussed in detail in Chapter 2. The aim of this thesis is to provide an alternative implementation that avoids these problems while preserving the *ACI(D)* properties.

## Chapter 2

# Introduction

## 2.1 Software Transactional Memory

Software Transactional Memory (STM in the following) is a programming language independent synchronization concept. Today STM is implemented in all common programming languages<sup>1</sup>. To understand the benefits of STM, take a look at the following example:

```
type Account = MVar Int

transfer :: Account -> Account -> Int -> IO ()
transfer src dst am = do
    balSrc <- takeMVar src
    balDst <- takeMVar dst
    putMVar src (balSrc - am)
    putMVar dst (balDst + am)
```

This is a simple implementation of a bank account and an associated transfer function. This implementation uses an `MVar` for synchronization. An `MVar` is a buffer with a capacity of one. This buffer can either be empty or filled. If the `MVar` is empty, every `takeMVar` operation on this `MVar` blocks until it is filled. If the `MVar` is filled, `takeMVar` empties the `MVar` and return the value. `putMVar` is the opposite operation. It fills the `MVar` with a value, if it is empty and suspends if the `MVar` is already filled.

This means `transfer` first empties both `Accounts`, then modifies the balances and at last writes back the new balances. At first glance this function seems to work fine, but the following example contains a deadlock:

Thread 1:

```
main = do
    transfer acc1 acc2 50
```

Thread 2:

```
main = do
    transfer acc2 ac1 50
```

The problem is the mutual access of the `MVars`. If both threads take their `src` at the same time, they will both wait for `dst`<sup>2</sup>. To avoid this deadlock we can rewrite the code:

```
transfer src dst am = do
    srcBal <- takeMVar src
    putMVar src (srcBal - am)
    dstBal <- takeMVar dst
    putMVar dst (dstBal + am)
```

<sup>1</sup>Even though STM is language independent, I will present the STM library in Haskell since this thesis is about STM in Haskell

<sup>2</sup>In fact is `transfer acc1 acc1 50` enough to evoke a deadlock

This indeed solves the problem regarding the deadlock. In return we lose consistency. For a brief moment we see an inconsistent state. Since the amount was already subtracted from one account, but not yet added to another account. This inconsistent state is observable by other threads. This was not possible in the first implementation.

We can use STM to avoid these problems. STM provides a single element buffer named `TVar`. A `TVar` always holds a value and is never empty. `TVars` are read and written with the functions `readTVar` and `writeTVar`, respectively. In contrast to `putMVar` and `takeMVar`, the `TVar` operations are not `IO` actions but `STM` action. `STM` is an instance of `Monad`, hence multiple `STM` actions can be combined using the comfortable `do`-notation. The following code represents the example from above implemented with `TVars` rather than `MVars`:

```
type Account = TVar Int

transfer :: Account -> Account -> Int -> STM ()
transfer src dst am = do
  srcBal <- readTVar src
  dstBal <- readTVar dst
  writeTVar src (srcBal - am)
  writeTVar dst (dstBal + am)
```

Note the type of `transfer` is no longer an `IO` action, but an `STM` action. Besides this the code looks similar to the `MVar` version.

In order to execute a transaction the function `atomically :: STM a -> IO a` is used. Since `readTVar` and `writeTVar` do not lock the `TVar`, the following example evokes no deadlock:

Thread 1:

```
main = do
  atomically $
    transfer acc1 acc2 50
```

Thread 2:

```
main = do
  atomically $
    transfer acc2 acc1 50
```

Furthermore ensures STM the *ACID* properties. The *ACID* properties were introduced in (Gray and Reuter, 1992). In the case of software transactions the *ACID* properties mean the following:

- *Atomicity*: the transaction executes all operations or none.
- *Consistency*: all modifications of a transaction are committed at the same time. No transition state is observable.
- *Isolation*: no concurrency is observable by a transaction. Each transaction can work as if it is the only transaction.
- *Durability*: ensures the perseverance of the changes. In the case of software transactions this is not necessary.

These properties explain the name `atomically`, because the enclosed code seems to be executed instantaneously without any interactions with other threads.

Before we turn over to the implementation of STM, we take a deeper look at the interface of the STM. `newTVar :: a -> STM (TVar a)` creates a new `TVar`. Since a `TVar` always holds a value, an initial value has to be passed to create a `TVar`. There is no function like `newEmptyTVar`.

Besides functions to create and access TVars, there are functions to alter the control flow. `retry :: STM a` is a generic STM action that indicates a failure, thus whenever a transaction engages a `retry` it restarts. The transaction is **not** restarted immediately. The transaction restarts, if at least one of the TVars it has read is modified. If the transaction would restart immediately (and no TVar has changed), the transaction would run into the same `retry` again.

With `orElse :: STM a → STM a → STM a` you are able to express alternatives. `orElse` executes the first transaction and ignores the second transaction, if the first transaction is successful. If the first transaction fails (retries), the second transaction is executed instead of the first one.

Note that it is not possible to execute IO action withing a transaction, which means that not side effects can occur. Furthermore this means restarting a transaction will never lead to the reexecution of irreversible operations. The reason is that the computations of transactions are done within the STM Monad. In other words the type system of Haskell forces us to write correct transactions.

## 2.2 Implementation

I will now give an overview on the current implementation of STM in Haskell. Afterwards I will analyze the problems of this implementation.

Even though the current implementation uses a low level C-library, I will retain an abstract view on the implementation, since the technical details do not matter for the course of this thesis. I will present an abstract view on the implementation to understand how the ACID properties are ensured.

The execution of a transaction is split in two phases. First the computation phase and second the commit phase.

### 2.2.1 Computation Phase

In this phase the user written operations are evaluated. The `writeTVar` operations do not write the original TVars immediately to preseve the ACID properties. Instead the so called *writeSet* is used. The *writeSet* stores pairs of TVars and values. The *writeSet* fulfills two purposes. First, if a transactions finishes and is found to be valid it needs to commit its changes to the original TVars. These changes are logged in the *writeSet*. Second, the *writeSet* serves as a local cache. If the user writes a TVar, the TVar and the associated value are entered in the *writeSet*, not in the original TVar. If the user reads this TVar afterwards, he would not be able to see his own modifications, since the original TVar is unchanged. To see his own modifications every `readTVar` first looks up the value in the *writeSet* and returns the associated value if present. Only when the TVar is not present in the *writeSet*, the original TVar is read. The programmer could keep track of the values he has written himself and thus would not need to search these values in the *writeSet*, but this would hinder the composability, which was an important motivation for STM.

Additionally a *readSet* is created for every transactions. This *readSet* hold pairs of TVars and version numbers. These version numbers are needed in the commit phase, which is explained in the next section.

Take a look at the following example:

```
transaction = do
  a <- readTVar t1
  b <- readTVar t2
```

```
writeTVar t1 b
writeTVar t2 a
```

This code would lead to the following read and writeSet:

```
readSet  = {(t1,0),(t2,0)}
writeSet = {(t1,b),(t2,a)}
```

Note that the values in the readSet are arbitrarily chosen. The exact value is not important, it just needs to be updated to a new unique value everytime it is written by a successful transaction. In this case an increment operation is enough to create a new unique value (if we ignore the overflow problem).

`newTVar` creates a new TVar and initializes this TVar. Afterwards this TVar can be used just like already existing TVars. Even if the transaction is rolled back, the new created TVars are not deleted explicitly. This work is done by the garbage collector, since the TVars are not further referenced, if the transaction which created them is rolled back.

`retry` aborts the computation and returns a result that indicates a failure. This result may be intercepted by `orElse` or is passed to `atomically` directly.

If `atomically` receives a result that indicates a failure, it aborts the transaction and restarts the transaction as soon as at least one of the TVars in the readSet has changed. These changes can be checked by comparing the version numbers from the readSet with the version numbers from the original TVars.

`orElse` on the other hand reacts differently on the result that indicates a failure. Remember the type of `orElse` :: STM a -> STM a -> STM a. Consider the following example:

```
transaction = do
  trans1
  trans2 'orElse' trans3
```

This transaction first executes `trans1`. Before `trans2` is executed the current writeSet is copied (name it `ws1` for now). Afterwards `trans2` is executed. If the execution leads to `retry`, the writeSet is set to `ws1` and `trans3` is executed<sup>3</sup>. The readSet remains unchanged, thus the TVars added by `trans2` are included. If the execution of `trans2` leads not to `retry`, the result, the readSet, and the writeSet produced by `trans2` are the result of `transaction`. In that case `trans3` is ignored.

It is crucial that the readSet produced by `trans2` is preserved. Otherwise the TVars read by `trans2` may be changed and `transaction` would not notice this. This would contradict the isolation of the ACID properties.

In conclusion the interface functions of STM are processed as follows in the computation phase:

- `writeTVar`: Add TVar and value to writeSet
- `readTVar`: Lookup the value in writeSet, if not present read the original TVar and add TVar and version number to the readSet.
- `newTVar`: Create and initialize a new TVar.
- `retry`: Return a result that indicates a failure.
- `orElse`: Backup the writeSet, execute the first transaction, if it fails restore the writeSet and execute the second transaction, else return the results of the first transaction.

---

<sup>3</sup>This means the writes evoked by `trans2` until then are discarded.

## Appendix A

# Frequently Asked Questions

### A.1 How do I change the colors of links?

The color of links can be changed to your liking using:

```
\hypersetup{urlcolor=red}, or  
\hypersetup{citecolor=green}, or  
\hypersetup{allcolor=blue}.
```

If you want to completely hide the links, you can use:

```
\hypersetup{allcolors=.}, or even better:  
\hypersetup{hidelinks}.
```

If you want to have obvious links in the PDF but not the printed text, use:

```
\hypersetup{colorlinks=false}.
```





# Bibliography

- Dijkstra, Edsger Wybe (1965). "Cooperating Sequential Processes, Technical Report EWD-123". In:
- Gray, Jim and Andreas Reuter (1992). *Transaction processing: concepts and techniques*. Elsevier.
- Harris, Tim et al. (2005). "Composable Memory Transactions". In: *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP '05. Chicago, IL, USA: ACM, pp. 48–60. ISBN: 1-59593-080-9. DOI: 10.1145/1065944.1065952. URL: <http://doi.acm.org/10.1145/1065944.1065952>.