

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

December 12, 2016

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, December 12, 2016

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

Declaration of Authorship	iii
Abstract	v
Acknowledgements	vii
1 Motivation	1
2 Introduction	3
2.1 Software Transactional Memory	3
A Frequently Asked Questions	5
A.1 How do I change the colors of links?	5
Bibliography	7

List of Figures

List of Tables

List of Abbreviations

LAH	List Abbreviations Here
WSF	What (it) Stands For
STM	Software Transactional Memory
ACID	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 deadlock¹. 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

¹In 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 a 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 the following 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 languages¹. 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`². 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)
```

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

²In 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 `STM` 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.

In order to execute a transaction the function `atomically :: STM a -> IO a` is used. The following example evokes no deadlock:

Thread 1:

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

Thread 2:

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

The `STM` library ensures the *ACID* properties, when executing a transaction. The *ACID* properties were introduced in (Gray and Reuter, 1992). In the case of Haskell 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. This is in the case of software transactions not necessary.

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>.