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 13, 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 13, 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 **7**
A.1 How do I change the colors of links? 7

Bibliography **9**

List of Figures

List of Tables

List of Abbreviations

LAH	**L**ist **A**bbreviations **H**ere
WSF	**W**hat (it) **S**tands **F**or
STM	**S**oftware **T**ransactional **M**emory
ACID	**A**tomicity **C**onsistency **I**solation **D**urability

Chapter 1

Motivation

Modern computer architecture includes multicore processors. To utilize these multicore 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. Aditionally there are technical challenges. A new schedular is needed and hardware accesses (Printer, Display, etc.) need to be sequential for example. These are challenges the operating system usually handles. The 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 thrid 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 inteded 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 a abstract datatype which holds an Interger 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 suspenden, 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[1]. This means there is a schedule, where no progress of the systen 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 verry hard even when using one or few semaphores. It is nearly impossible to avoid deadlocks when you trie 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). Sofware transactions are inspired by the long known database transactions (Gray

[1]In the course of this thesis I will refer to deadlocks as a static propertie 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 independend synchronization concept. Today STM is implemented in all common programming languages[1]. 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)
```

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

[2]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 allready 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 an 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. Besided this the code locks 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 they access, the following exmaple evokes no deadlock:

Thread 1:

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

Thread 2:

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

Furthermore does STM library ensures the *ACID* properties. 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. In the case of software transactions this is not necessary.

These properties explain the name `atomically`.

Before we turn over to the implementation of STM, we take a look at the rest of the STM interface. `newTVar :: a -> STM (TVar a)` creates a newTVar. 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 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 frist 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.

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.