

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

February 6, 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, February 6, 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

| | |
|--|------------|
| Declaration of Authorship | iii |
| Abstract | v |
| Acknowledgements | vii |
| 1 Motivation | 1 |
| 2 Introduction | 3 |
| 2.1 Software Transactional Memory | 3 |
| 2.2 Implementation | 6 |
| 2.2.1 Computation Phase | 6 |
| 2.2.2 Commit Phase | 8 |
| 2.2.3 Notes on the Implementation | 9 |
| 2.3 Problems | 10 |
| 2.3.1 Unnecessary Rollback | 10 |
| 2.3.2 Unnecessary Recomputations | 11 |
| 3 Concept | 13 |
| 3.1 Unnecessary Rollbacks | 13 |
| A Frequently Asked Questions | 15 |
| A.1 How do I change the colors of links? | 15 |
| Bibliography | 17 |

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. For example a new scheduler is needed and hardware accesses (Printer, Display, etc.) need to be coordinated. 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 back 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 non intended behaviour such as this 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, Marlow, et al., 2005). Software transactions are inspired by the long known database transactions

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

(Gray 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 available 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 acc1 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 is already subtracted from one account, but not yet added to the other account. This inconsistent state is observable by other threads. This is 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 actions³. 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` instead of `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. Apart from 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 contains no deadlock:

Thread 1:

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

Thread 2:

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

This is because STM ensures the *ACID* properties. The *ACID* properties were introduced in (Gray and Reuter, 1992) for database transactions. These properties were adapted for software transactions later on. 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 appears 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.

³If you are wondering when I use `SMT` and when `STM`. I use `STM` when I refer to the Haskell type constructor and `STM` when I refer to `STM` as library

`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 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 within a transaction, which means that not side effects can occur. Furthermore this means restarting a transaction will never lead to the re-execution 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.

For single threaded programming, abstraction and composability are key features. These features allow us to combine smaller pieces of code into more complex pieces of code. These feature are not available for lock based concurrent programming. Compose correct lock based concurrent functions most likely lead to deadlocks or inconsistencies. Consider the following example:

```
withdraw :: Account -> Int -> IO()
withdraw acc am = do
    bal <- takeMVar acc
    putMVar acc (bal - am)
```

```
deposit :: Account -> Int -> IO()
deposit acc am = withdraw acc (-am)
```

These are functions to withdraw and deposit money from and on an account. The natural way to implement `transfer` is:

```
transfer :: Account -> Account -> Int -> IO()
transfer src dst am = do
    withdraw src am
    deposit dst am
```

We reuse the functions that are already defined instead of coding everything from the scratch. In our example this is equivalent to the solution suggested above to eliminate the deadlock. This implementation is free of deadlocks, but it lacks consistency. Thus building complex concurrent operations can not take advantage of abstraction and composability. We always need to code everything from the scratch. This is error prone in compared to the step wise combination of smaller operations into bigger operations.

`STM` allows us to use this important programming paradigm for concurrent programming. Thus the following example provides deadlock freedom as well as consistency.

```
withdraw :: Account -> Int -> STM()
withdraw acc am = do
    bal <- readTVar acc
```

```

writeTVar acc (bal - am)

deposit :: Account -> Int -> STM()
deposit acc am = withdraw acc (-am)

transfer :: Account -> Account -> Int -> STM()
transfer src dst am = do
    withdraw src am
    deposit dst am

```

We can combine arbitrary transactions to more complex transactions while preserving the ACI properties. This greatly benefits the readability of the code. In addition it increases the efficiency of the development process, because we are able to reuse code that we already found to be correct. This was also one of the main motivations of paper (Harris, Marlow, et al., 2005) which forms the foundation of STM in Haskell.

2.2 Implementation

I will now give an overview on the current implementation of STM in Haskell. For a detailed description of the implementation and the theory behind it refer to (Harris, Marlow, et al., 2005). Afterwards I will analyze the problems with 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 are not important for the course of this thesis. I will present an abstract view on the implementation to understand how the ACI properties are ensured.

The execution of a transaction (a call of `atomically`) is split in two phases. First the computation phase and second the commit phase.

2.2.1 Computation Phase

In this phase the operations that were written by the user are evaluated. To preserve the ACI properties, the `writeTVar` operations do not write the actual TVars immediately. Instead a data structure called *writeSet* is used. The *writeSet* stores pairs of TVar and value. The *writeSet* fulfills two purposes. First, if a transaction finishes and is found to be valid it needs to commit its changes to the actual TVars. These changes are logged in the *writeSet*. Second, the *writeSet* serves as a local cache. If the user writes an TVar, the TVar and the associated value are entered into the *writeSet*, not into the actual TVar. If the user reads this TVar afterwards, he would not be able to see his own modifications, since the actual 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 actual 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 is an important motivation for STM.

The TVar not only holds a value, but also a version number, which is updated everytime the TVar is modified. Additionally a *readSet* is created for every transaction. This *readSet* holds 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 readSet and writeSet:

```

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

```

Note that the version numbers in the readSet are arbitrarily chosen for this example. The exact value is not important, it only needs to be updated to a new unique value every time 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 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 that created them is rolled back.

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

If `atomically` receives an result that indicates an failure, it aborts the transactions 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 actual TVars.

`orElse` on the other hand reacts differently on the 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 executed⁴. 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 ACI 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 actual 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.

⁴This means the writes evoked by `trans2` until then are discarded.

- `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.

2.2.2 Commit Phase

After the `readSet` and `writeSet` are calculated and no further STM actions need to be processed, the commit phase starts. At first the transaction locks all TVars in its `readSet` in order to avoid concurrency problems at this point. If a transaction tries to acquire a lock that is already locked, it releases all its locks restarts the commit phase to avoid deadlocks.

Then the transactions validates. Remember that the `readSet` is a collection of pairs consisting of a TVar and a version number. It is sufficient to compare the version numbers stored in the `readSet` with the version numbers in the actual TVars to validate. If all version numbers match, it means the transactions has seen a consistent state of the system, because no TVar has changed after it was read by the transaction. Thus the transaction is valid. If there is at least one miss match, it means a TVar was changed after the transaction read it. Thus the isolation is violated and the transaction is invalid.

There are two possible results of the computation phase. First, the result of the computation phase indicates a failure (i.e. engaged a `retry`). If the transaction is invalid, it is rolled back. This means the `readSet` and `writeSet` are discarded and the transaction is restarted. If the transaction is valid, the TVars in the `readSet` are observed. As soon as one of these TVars is modified, the transaction is rolled back.

Second, the computation phase returns a result that indicates a success. Depending on the result of the validation the transaction is further processed. If the transaction is valid, additionally the TVars in the `writeSet` are locked (the TVars in the `readSet` are not unlocked yet). After that the actual commit is processed. The values stored in the `writeSet` are written to the actual TVars. The version numbers of the written TVars are updated as well. The last step of the transaction is to unlock the TVars and return the result of the transaction to the caller of `atomically`.

If the transaction is found to be invalid, the transaction is rolled back after the locks of the TVars has been released.

Lets take a look at the following example:

```
transaction1 = do
  a <- readTVar t1
  writeTVar t1 (a-1)

transaction2 = do
  a <- readTVar t1
  writeTVar t1 (a+1)
```

Assume the transactions are executed parallel and the initial version number of `t1` is 0 and the initial value of `t1` is 5. If `transaction1` first executes its computation phase, its `readSet` contains `(t1, 0)`. If now `transaction2` is executed and finishes the computation phase before `transaction1` enter the commit phase, its `readSet` contains `(t1, 0)`. Since the TVars are locked before the validation starts, only one of the two transactions could validate at a time.

Assume `transaction1` validates first and sees that the version number of `t1` is still 0, hence it is valid. Then `transaction1` commits, which is to publish its `writeSet`. The `writeSet` contains `(t1, 4)`. Consequently 4 is written to `t1`. Furthermore is

the version number of `t1` updated to 1. In the end `t1` is unlocked (and `()` is return, because it is the result of `writeTVar t1 (a-1)`).

Since `t1` is unlocked `transaction2` can validate. Now the version number of `t1` is 1, but the version number in the `readSet` of `transaction2` is 0. This means the validation fails and the transactions is rolled back. After the reexecution of the computation phase the `readSet` would contain `(t1, 1)` and the `writeSet` `(t1, 5)`. The validation succeeds and the `writeSet` is published. The result is the intended one. After executing an atomic increment and an atomic decrement, the TVar value is the same as the initial value.

If there were no validation, `transaction2` would also commit at this point in the first try. Thus the value of `t1` would be set to 6. This would violate the ACI properties. Assuming that the decrement and increment operations are atomic operation, the value of a TVar should not change if it is incremented and decremented once.

2.2.3 Notes on the Implementation

Larus and Rajwar describe in their book (Larus and Rajwar, 2007, Chapter 2) different design options to be done when implementing a (Software) Transactional Memory. While most of these options effect the performance of a system, some also effect the semantics of the system. I will now discuss the design options that are important for this thesis ⁵.

Deferred and Direct Updates. The way a STM system modifies the underlying data structure can either be *deferred* or *direct*. Direct updating systems are writing the actual objects when a write operation is called. In the case of Haskell this would mean, every time `writeTVar` is called. Deferred updating systems on the other hand buffer the write operations to commit them later on. Haskell STM is a deferred updating system, since the values are buffered in the `writeSet` before they are committed. This design options does not effect the semantics of the system. While a direct system loses performance, when transaction is rolled back, because the initial values need to be restored, a deferred systems contains an overhead due to the `writeSet` and the need to look up values in it when an Object is read. Neither mechanism is better than the other; it depends on the application STM is used in. (Harris, Plesko, et al., 2006) compares a deferred and a direct system. They show that the performance of a direct update system is significantly higher than that of a deferred system, when the reads outnumber the writes by far.

Early and Late Conflict Detection. A STM system needs to detect conflicts in order to ensure the ACI properties. This can be done as soon as the conflict occurs or at some point later before the transaction commits. If the system uses a late conflict detection, transaction may work on an inconsistent state. This may leads to loops or exceptions. So this a semantic relevant design decision. Haskell STM uses a late conflict detection. By validating the `readSet` before comitting the transaction a possible conflict is detected. This implies the transaction may work on an inconsistent state until it attempts to commit. this means the transaction may run into an infinite loop, because it saw an inconsistent state. To avoid this problem, additional validations are performed each time the executing thread yields. In conclusion the transaction may see an inconsistent state and run into an infinite loop because of that, but will we restarted eventually due to validations performed in the middle of the transaction.

⁵The names used in the following part are taken from (Larus and Rajwar, 2007, Chapter 2)

2.3 Problems

I will now explain two problems with this implementation. These problems can be examined independently. The first problem is about *when* a transaction is rolled back and the second problem is about *how* a transaction is rolled back.

2.3.1 Unnecessary Rollback

Remember the STM implementation of `transfer` and its example use given in 2.1:

```
transaction1 = do
  atomically $
    transfer acc1 acc2 50
```

```
transcation2 = do
  atomically $
    transfer acc2 acc1 50
```

The implmentation is correct, but not verry efficient in this case. Take a look at the inlined functions to understand the problem:

```
transaction1 = do
  a1 <- readTVar acc1
  a2 <- readTVar acc2
  writeTVar acc1 (a1 - 50)
  writeTVar acc2 (a2 + 50)
```

```
transaction2 = do
  a1 <- readTVar acc2
  a2 <- readTVar acc1
  writeTVar acc2 (a1 - 50)
  writeTVar acc1 (a2 + 50)
```

Due to the scheduler the thread can run in a sequential order. This case may occur, but is not desirable. It means there is no performce improvement by executing this on a multiple cores/processors. Thus the efforts to use multiple threads are futile in the first place. This is not a problem specific to STM, but to all synchronization mechanisms. If the resulting multi thread program is not scheduled in a way that it is executed parallel, these mechanisms are a performance deterioration rather than a performance improvement. Since we cannot access the scheduler, we ignore this case.

The second case is that these transactions are run parallel. This should be the better case, because the implementation has a chance to improve the performance. Sadly this is not the case. To understand why, we need to take a close look at the execution. Let us assume both threads execute their computation phase parallel. This means both read the initial values of `acc1` and `acc2` and add these information to their `readSet`. Furthermore add both transactions entries for `acc1` and `acc2` to their `writeSet`. Then both transactions try to commit, thus try to lock the TVars. If `transaction1` gets the locks for `acc1` and `acc2` it validates and commits, since none of the TVars has changed after `transaction1` read them. After that `transaction2` aquires the locks and validates. Since `transaction1` changed the version numbers of `acc1` and `acc2` by committing, the validation fails and `transaction2` is restarted. In conclusion no performance improvement was achieved. Both threads are still executed on after another and not parallel as intended.

If we rearrange the operations of `transfer`, we are able to see how this can be improved. Note that we can rearrange the operations to a certain degree without changing the semantics of the resulting code due to the ACI properties.

```
transfer src dst am = do
  srcBal <- readTVar src
  writeTVar src (srcBal - am)
  dstBal <- readTVar dst
```

```
writeTVar dst (dstBal - am)
```

Transfer basically consists of two parts. Decreasing the balance of the source account and increasing the balance of the destination account. The actual values of `src` and `dst` are not important for these transactions. If we delay the evaluation of `readTVar` to the commit phase, we avoid the aforementioned **unnecessary** rollback, because no transaction would read a value, that is overwritten by another transaction afterwards.

If we refer to our example:

Thread 1:

```
transaction1 = do
  a1 <- readTVar acc1
  a2 <- readTVar acc2
  writeTVar acc1 (a1 - 50)
  writeTVar acc2 (a2 + 50)
```

Thread 2:

```
transaction2 = do
  a1 <- readTVar acc2
  a2 <- readTVar acc1
  writeTVar acc2 (a1 - 50)
  writeTVar acc1 (a2 + 50)
```

If we change the semantics of `readTVar` by delaying the evaluation, the following happens. Both transactions will execute the computation phase simultaneously. This means for `transaction1` adding `(acc1, (a1 - 50))` and `(acc2, (a2 + 50))` to their `writeSet` (this is analog for `transaction2`). At the first glance this seems to be incorrect since the value of `a1` and `a2` are not yet present. For Haskell this is quite common. Haskell is a non-strict language, which means passing unevaluated expressions is normal.

After the computation phase the commit phase follows. The first step is to lock the read TVars in order to perform the validation. Since both transactions used the same TVars, they will commit successively instead of parallel.

Assume `transaction1` gets the locks first and tries to validate⁶. Since the `readSet` is empty the validation is unnecessary; the empty `readSet` is always valid. Before `transaction1` is able to commit its `writeSet` it needs to evaluate `a1` and `a2`. By doing this the `readTVar` operations are evaluated. Hence the TVar and its version number is added to the `readSet` and the actual value is returned. Since the `readSet` is already validated at this point it is superfluous.

After `transaction1` finished and released the locks, `transaction2` acquires these locks and validates. The `readSet` of `transaction2` is also empty, thus the validation succeeds. Then `transaction2` evaluates the `readTVar` operations and commits its `writeSet` before releasing the locks.

Both transactions run parallel as far as possible and did not roll back. In Chapter 3 are the limitations of this idea presented and the challenges that arise when implementing it. Furthermore are solutions to these challenges introduced.

2.3.2 Unnecessary Recomputations

While the first problem dealt with the question *when* transactions should be rolled back, the second problem investigates the question *how* transactions are rolled back. Take a look at the following example:

```
transaction = do
  a <- readTVar t1
  writeTVar t1 (f a)
```

⁶You could argue that evaluating `readTVar` operation is necessary before validating, but this would not change the validity of the transaction, since the TVars are locked and can not be modified by other transactions at that point.

```
b <- readTVar t2
writeTVar t2 (g b)
```

This transaction contains two independent statements. The first two lines of the transaction form the first statement. This is independent of the last two lines. Independent means their side effects or results do not influence each other. While the first line influences the second line, it does not influence the last two lines and vice versa.

If the transaction is now executed it computes its writeSet and readSet first. Then it locks the TVars and validates. The validation fails if either of the TVar has changed after it was read by the transaction. If the validation fails the transaction is rolled back. Which means the readSet and writeSet are discarded, regardless which TVar was the reason for the failed validation.

For example if t_1 was modified after the transaction read it, the transaction is rolled back and both statements are executed again. This includes the read and write of t_2 although the value of t_2 did not change. Hence the exact same code with same inputs and the same (relevant) environment is executed twice. If we are able to invalidate parts of transactions instead of transactions as a whole, we can save time when rolling back a transaction. In the previous example we can use the information that only t_1 changed. If we roll back this transaction, we only need to execute the first two actions, because the last two actions do not depend on t_1 .

This concludes the overview on the problems of the current STM implementation. We will now turn over to the solution for this problems.

Chapter 3

Concept

I will now explain how I engaged the previously stated problem.

3.1 Unnecessary Rollbacks

Remember the idea given in 2.3.1. I suggested to delay the evaluation of `readTVar` operations to the commit phase rather than doing them directly in the computation phase. **This part could be the first point of the fix chapter** While the idea would work for this example, the idea would not work for the following example:

```
limitedTransfer src dst am = do
  srcBal <- readTVar src
  if a1 < am
    then return ()
    else do dstBal <- readTVar dst
            writeTVar src (srcBal - am)
            writeTVar dst (dstBal + am)
```

If we use this function the result of `readTVar src` is needed in the computation phase and therefore the evaluation cannot be delayed to the commit phase. The value is needed to decide on the condition of the if expression. To be exact the value is needed to determine the control flow.

This leads to the question whether there is a way to determine if the result of a `readTVar` effects the control flow or not. The current implementation is we are not able to do this. The problem is the bind operator: $\gg= :: STM\ a \rightarrow (a \rightarrow STM\ b) \rightarrow STM\ b$ ¹. This operator allows us to extract the result of an STM action (for example a `readTVar`) from the STM context. This means the STM library loses any possibility to observe this value. Thus the library is not able to decide if the value is used to alter the control flow.

If the library handles a value that is **not** used for branch conditions as if it were used for branch conditions, it may lose performance, but preserves the correct semantics. If the library on the other hand handles a value that is used for branch conditions as if it were not, the library would not perform unnecessary rollbacks, but may violate the ACID properties. Thus the only way to ensure the correctness of the implementation is to handle all values as control flow critical values.

¹Remember that the `do` notation used so far is syntactic sugar for $\gg=$ and \gg

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, Simon Marlow, 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>.
- Harris, Tim, Mark Plesko, et al. (2006). "Optimizing memory transactions". In: *ACM SIGPLAN Notices* 41.6, pp. 14–25.
- Larus, James R and Ravi Rajwar (2007). *Transactional memory*. Vol. 1. 1. Morgan & Claypool Publishers.