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

March 2, 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, March 2, 2017

_____

Christian-Albrechts-University

Abstract

Faculty of Engineering
Department of Computer Science

Master of Science

Alternative Software Transaction Implementation in Haskell

by Lasse Folger

TODOTODOTODOTODOTODOTODOTODOTODO
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

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
TVar	**T**ransactional **Var**iable

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. 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. 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 problems 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 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 inteded behaviour is called *lost update*. The efforts to avoid non intended behaviour such as this are called synchronization.

Although 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 suspended, because it called `P` on the semaphore, that thread awakens. After the thread has awaken, 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 very 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 in Haskell the so called software transactions were introduced (Harris, Marlow, et al., 2005). Sofware transactions are inspired by the long known database

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

transactions (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. If multiple thread work on the same data they do not influence each other indirectly. The only way threads can influence each other is by communicating through shared memory. **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). The STM library provides an interface which allows the user to process arbitrary operation on one element buffers (so called TVars). The operations can be grouped to *transactions*. When a transaction is executed the library ensure the ACI(D) properties. This is done by optimistically executing the transaction. If a conflict is detected, the changes of the transaction are discarded and the transaction is restarted (also 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 languages. Since the subject of this thesis is Haskell, we will not investigate STM in general. 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` [1]. 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] 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 is allready withdrawn from one account, but not yet deposited on 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 both of these problems. STM provides a single element buffer named `TVar`. In contrast to an MVar, 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[2]. `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 an STM action (called transaction in the following) the function `atomically :: STM a -> IO a` is used. The following example contains no deadlock, because `readTVar` and `writeTVar` do not lock the TVar:

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*: all operations of the transaction are executed 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 any transaction. transactions do not influence each other indirectly.
- *Durability*: ensures the perseverance of the changes.

In the case of software transactions the *Durability* is not demanden, which is why we will refer to the ACI properties in the rest of the thesis.

These properties explain the name `atomically`, because the enclosed code appears to be executed instantaneously without any interactions with other transactions. Before we turn over to the implementation of STM, we take a deeper look at the interface of the STM.

[2]If 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.

Note that it is not possible to execute `IO` action within a transaction, which means that no 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. Composing correct lock based concurrent functions often leads 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 comparison to the step wise combination of smaller operations into more complex 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 was already found to be correct. This was also one of the main motivations of the paper (Harris, Marlow, et al., 2005) which forms the foundation of STM in Haskell.

The reason this works is because always the whole transaction (`STM` action) is considered as one block for which the ACI properties must hold. Thus the user marks the critical section by defining them as one transaction and the library ensures the correctness and deadlock freedom. The user only needs to think about which actions need to be processed together. This is comparable to a lock based version with a single lock[3]. Everytime the user wants to process a critical section he takes the lock before this section and releases the lock afterwards. Then the critical section are processed isolated and problem such as race conditions and lost updates does not occur. The performance on the other hand is devestating and does not scale well, because all critical sections are sequentialized. This is for most modern systems not acceptable, thus this solution is not feasible.

2.2 Implementation

In this section we explore the current implementation of STM in Haskell, more specific in GHC. For a detailed description of the implementation refer to `https://ghc.haskell.org/trac/ghc/wiki/Commentary/Rts/STM`.

Even though the current implementation uses a low level C-library, we retain an abstract view on the implementation, since the technical details are not important for the course of this thesis. The implementation is outlined to understand how the ACI properties are guaranteed.

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

Each transaction holds a log for the TVars it has accessed. The log contains four elements per entry. These are:

- tvar
- expectedValue
- newValue

[3]For now we assume that if the user tries to acquire the lock, when he already it, it is a NOOP.

- versionNumber

The `versionNumber` is only used to prevent a very subtle bug and thus not considered in this thesis. The log is extended and modified by the transactional operations `writeTVar` and `readTVar`. `newTVar` on the other hand creates the new TVar directly. Whenever `readTVar` is called the associated TVar is lookep up in the log. If it is present, the `newValue` is returned. If it is not present, a new entry in the log is created. While `tvar` is the passed TVar, `newValue` and `expectedValue` are the actual value of the TVar and `versionNumber` is the actual verison number. This is one of the two times in the computation phase when the transaction accesses the actual mutable data structures. After the entry is created and added to the log, the actual value is returned.

A call of `writeTVar` also looks up the associated TVar in the log. If it is present, the field `newValue` is set to the value passed to writeTVar. If it is not present, a new entry is created. The `tvar` is the passed TVar and the `newValue` is the passed value and `expectedValue` is the actual value of that TVar. This is the other time the actual mutable data structures are accessed in the computation phase. To enter the actucal value of the TVar at this point is not needed to preserve the ACI properties. This is done to simplify the implementation at the risk of an additional rollback. On the other hand it is unusual to write a TVar that was not read before, because this means to overwrite the value of a TVar and thus discarding the initial content of that TVar.

This log fulfills two purposes. One purpose of the log is the use in the commit phase which is described in Section 2.2.2. The other is the interaction between `readTVar` and `writeTVar`. The `readTVar` operations are able to see the results of preceding `writeTVar` operations in the log. Without the log `writeTVar` would need to access the actual TVar. This on the other hand would imply that other transactions would be able to see inconsistent intermediate states of the system; a violation of the ACI properties. It may seem unnecessary to read a TVar that the transaction itself wrote before. The transaction should know what it writes and thus does not need to access such TVars. Nevertheless there are two reasons to allow it. The current implementation allow the user to combine all transactional actions in an arbitrary manner and the library ensures (at compile time) that it works correctly. To restrict the user to only read TVars he has not yet written, no longer allows the library to give this kind of guarantee at compile time; this contradicts the design concept of Haskell. The second reason is one of the core motivations of STM in Haskell: composability. With the restriction it is not possible to combine arbitrary correct STM functions to new more complex STM functions.

To understand how a log could look like, 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 log:

```
log = {(t1,a,b),(t2,b,a)}
```

The log contains two entries, because the transaction accessed two TVars. The first part of the entry denotes the TVar, the second part the expected value and the last part is the new value. The first entry contains the information that `t1` held the value `a` when it was first read and the value `b` is the new value of it. `t2` held the value `b`

and the new value is `a`. Before we will examine the commit phase, we will look at the other operations of the STM interface.

`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.

`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. Aborting a transaction means to discard the log. Since no observable operations are performed in the computation phase, nothing has to be undone. As soon as at least one of the TVars in the log has changed, the transaction is restarted. If the transaction is restarted immediately and no TVar has changed the transaction would reach the same `retry` again. These changes can be checked by comparing the `expectedValue` in the log with the actual value in the TVar. To avoid busy waiting the thread do not repeatedly check if the value has changed. The TVar has a queue for wait waiting threads. Each time a transaction successfully commits and writes a TVar it also checks if there is someone waiting in this queue. The committing thread then notifies all waiting threads.

`orElse` on the other hand reacts differently on the the result that indicates a failure. The implementation works with nested transactions, but to explain this in detail would go beyond the scope of this thesis. Nested transactions are not able to publish their writes on their own. When a nested transaction successfully commits(we see in the next section what this means), its log is integrated in the log of the surrounding transaction. Integrated means the logs are merged and in the case that there is a entry in both logs for one TVar, the entry of the outer transaction is discarded. If the nested transaction fails, because `retry` occurred and it is the first transaction of `orElse`, the log of the inner transaction is integrated in the log of the surrounding transaction, but the `newValue` fields of the inner log are ignored. If the nested transaction fails to validate the outermost transaction is rolled back.

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

- `writeTVar`: Look up TVar in log. If present update `newValue`. If not present read actual TVar and create new entry.
- `readTVar`: Look up TVar in log. If present return `newValue`. If not present read actual TVar and create new entry.
- `newTVar`: Create and initialize a new TVar.
- `retry`: Return a result that indicates a failure.
- `orElse`: Create a nested Transaction and reacts on the return value of that transaction.

2.2.2 Commit Phase

After the log is calculated and no further STM actions need to be processed, the commit phase starts. At first the transaction checks if the values in its log are still correct by *validating* its log. Validation denotes the process to check if the expectedValues are equal to the actual values in the TVars. In other words for each entry in the log the transaction reads the actual TVar and compares the value with the `expectedValue`

in the log. If at least one of these values does not match, the transaction is considered *invalid*. If the validation returns the transaction is instantaneously rolled back, by discarding the log and restart it computation. If all values match the the transaction is considered *valid*. If the validation returns valid, each entry in the log is processed.

If `expectedValue` differs from `newValue` the associated TVar is locked. The transaction has acquired all locks it needs, it validates again. This seems a bit wasteful in terms of resources, but locking the TVars is considered an expensive operation and thus the implementation tries to avoid this when ever possible. This process reduces the chance that the transaction acquires all locks and then finds out it is invalid and consequently a unnecessary locking of TVars. If the validation fails at this point, the transaction is rolled back after the locks has been released. If the transaction has acquired all locks and is valid the transaction is is ready to publish its changes. This means iterating on the log and update the actual TVars where `expectedValue` and `newValue` differ and simultaneously releasing the locks.

If the validation returns invalid it means at least one `expectedValue` is no longer correct. To roll back is essential to retain the ACI properties. The failed validation indicates that transaction has read an outdated value and possibly worked with this value. Take a look at the following example:

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

If this transaction is processed by two transactions in parallel. Both would read the initial value of `t1`, say `1`. So both would note in their log `(t1,1,1)`. After the `writeTVar` the log of both transactions would look contain `(t1,1,2)`. After that both transactions try to commit. Assume one transaction commits before the other transaction tries. [4] Then the transaction would find its log to be valid and lock `t1`. After that its log is still valid and so it modifies `t1` and releases the log. Then the second transaction tries to commit. Since the actual value hast changed to `2` it does no longer match the `expectedValue` and the transaction is rolled back. If the transaction would not be rolled back at this point and commit instead. The transaction would write `2` to the TVar (that already contains `2`). In the end this would means the value of `t1` is `2` after both transactions have finished. This is certainly not the intended behaviour after incrementing the TVar that holds `1` twice. This is the well known *lost update* problem.

By rolling back the second transaction it reads `t1` once more. The log contains `(t1,2,2)` after the `readTVar` operation and `(t1,2,3)` after the `writeTVar` operation. Then the transaction validates, locks, validates and finally publishes it modifications. In the end the value of `t1` is `3`; just as intended.

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 only the performance of a system, some also effect the semantics of the system. We will discuss in this section the design options that are important for this thesis [5].

[4] For simplicity we assume that no other transaction is running besides the two we are looking at.

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

Deferred and Direct Updates

The way a STM system modifies the underlying data structures 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 a transaction is rolled back, because the initial values need to be restored, a deferred systems contains an overhead due to the need to log values and looking them up. Neither mechanism is better than the other in general; it depends on the application that 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 reads outnumber 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 later before the transaction commits. If the system uses a late conflict detection, transactions may work on an inconsistent state. This may lead to loops or exceptions. So this design decision is relevant for the semantics. Haskell STM uses a late conflict detection. By validating the log 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. Exceptions raised by the transaction are handled like a `retry`. If the log is valid, the transcation waits until at least one TVar changed. If it is invalid the transaction is restarted immediately. In conclusion, the user of STM can not observe that the transaction worked on an inconsistent state.

Synchronization

The last important property of a STM system is the way it synchronizes transactions. In order to validate correctly the systems needs to make sure the validation result does not depend on race conditions and is correct until the commit is completed. This means either concurrent transactions are delayed or their commit does not change the each others validity. In Haskell the first approach is taken. When a transaction commits, the TVars in the log that are updated are locked, thus other transactions that may conflict are not able to commit at the same time. In order to avoid a deadlock, all locks are released and the transaction is rolled back when it tries to aquire the lock for a locked TVar. In the worst case this leads to the roll back of both transactions, however the chances are narrow. Rolling back the transaction seems to be harsh instead of waiting until the other transaction finishes and then trying to commit, but if two transactions try to lock the same TVar, both transactions try to write this TVar. This means at least one of the transactions is rolled back, since the TVar is logged with the old value. Thus the first transaction to commit would modify a value in the log of the other transcation.

2.3 Problems

In this section we turn over to the problems in the current 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 threads 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 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 threaded 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 in 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 at the same time. This means both read the initial values of `acc1` and `acc2` and add these information to their log. Furthermore add both transactions entries for `writeTVar acc1` and `writeTVar acc2` to their log. Then both transactions try to commit, thus try to lock the TVars. It is possible both transactions are rolled back at this point. Lets assume `transaction1` acquires the locks for `acc1` and `acc2`. Since no TVars were modified after `transaction1` read them, it validates and commits. If `transaction2` tries to access the TVars before `transaction1` has finished committing, it is rolled back. Thus it is possible for `transaction2` to read the old value once again. If `transaction2` is descheduled for the time `transaction1` commits, it is rolled back afterwards, because the values of `acc1` and `acc2` have were changed by `transaction1`. In conclusion no performance improvement was achieved. The most efficient execution is if both transactions are executed in a sequential order. As mentioned before, this not desirable for multithreaded programs.

This leads to two questions:

- When is it needed to roll back a transaction?

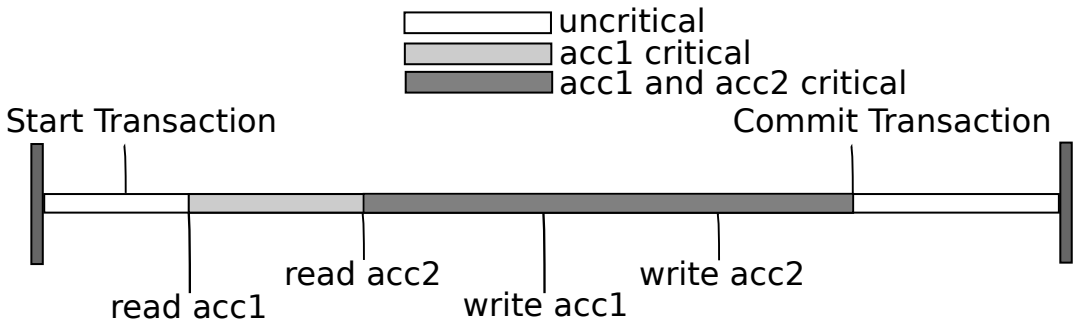

FIGURE 2.1: Time when the update of `acc1` or `acc2` causes a rollback.

- How can we avoid or at least decrease rollbacks?

A transaction needs to be rolled back if it is operating on data that is not a snapshot of the current memory. In other words if a value has changed after the transaction read this value. When a transaction reads a TVar, this TVar becomes *critical* for the transaction. Critical means a modifications of that TVar causes the transaction to roll back. Figure 2.1 visualizes when the TVars `acc1` and `acc2` are critical for `transaction1`. When `readTVar acc1` is executed the values becomes critical and stays critical until the transaction commits. If any other transaction commits a modification to `acc1` or `acc2`, while `acc1` and `acc2` are critical for `transaction1`, `transaction1` is rolled back to preserve the ACI properties. A solution other than the roll back is presented in 2.3.2.

This insight brings an intuitive way to deal with this problem. If we minimize the time the TVars are critical for a transaction we reduce the chance that this transaction is rolled back. If we rearrange the operations of `transfer`, we are able reduce the time `dst` is critical. 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)
```

With this implementation of transfer the time in that both TVars are critical is reduced. Figure 2.2 shows the effects of this for `transaction1`. The second TVar, namely `acc2`, is shorter critical than in the initial implementation. The time `acc1` is critical has not changed at all. Nevertheless it shows that delaying the execution of `readTVar` can reduce the time values are critical and by this the chance the transaction is rolled back. Our aim is to delay the execution of `readTVar` as far as possible to reduce to time that a TVar is critical for the transaction. We have already seen one option to achieve this; rearrange the operations of a transaction. This would require a kind of preprocessing in the compiling process, for example a source to source code transformation. The aim of this thesis is to provide an pure Haskell library. I do not intend to implement an extension to the compiler nor do I want to provide a source to source code transformer. The only other option is to alter implementation of `readTVar` and `writeTVar` without changing the *external* semantics of STM. External semantics are the semantics the user can observe and which effect the user written code.

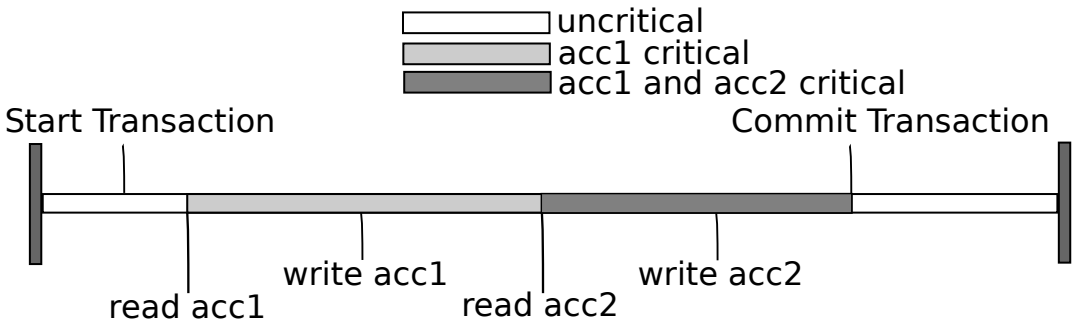

FIGURE 2.2: Effect of rearranging code with regards to the time `acc1` and `acc2` are critical for `transaction1`.

The critical time would be minimal if the TVars were read directly before or at the start of the commit phase. This would mean the chances that another transaction commits a change to a TVar that is critical are low or non existing. So the idea is to let the user define transactions like before, but changing the semantics of `readTVar` that it is evaluated in the commit phase. The user is able to define transactions like with the original implementation, but the delay of the evaluation shortens the time TVar are critical and thus the chance a transaction is rolled back.

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 `transaction1` adds `(acc1,a1,(a1 - 50))` and `(acc2,a2,(a2 + 50))` to its log (this is analog for `transaction2`). At the first glance this seems to be incorrect since the values of `a1` and `a2` are not yet present. For Haskell this is quite common. Haskells 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 validate[6]. Since the readSet contains an action whose result is the current value, the validation is unnecessary; it is always valid. To validate the log, `a` and `b` are evaluated. At last the new values are written to the TVars.

After `transaction1` finished and released the locks, `transaction2` aquires these locks and validates. The log of `transaction2` is also valid and also commits its changes.

[6]You 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.

Both transactions run parallel as far as possible and did not roll back. Chapter 3 presents the limitations of this idea and the challenges that arise when implementing it.

2.3.2 Unnecessary Recomputations

While the first problem dealed with then question *when* transactions need to be rolled back, the second problem investigates the question *how* transactions are rolled back. Lets take a look at our well known example:

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

This transaction contains two independent statements. The first two lines of the transacton 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 influces the second line, it does not influence the last two lines and vice versa.

If the transaction is executed, it computes its log first. Then it locks the TVars and validates[7]. The validation fails if either of the TVar has changed after it was read by the transaction. If the validation fails the transcation is rolled back. Which means the log is discarded, regardless which TVar was the reason for the failed validation.

Suppose a `transaction1` executes `transfer acc1 acc2 5` and is deschedules before committing. Then `transaction2` modifies `acc1` (and nothing else) and commits. This would cause `transaction1` to roll back and execute both parts of `transfer` again. This includes the read and write of `t2`, although the `t2` was not modified. Hence the exact same code with same inputs and the same (relevant) environment is executed twice. If we just execute the parts of a transaction that are invalid instead of all, we can save a considerable amount of time when a transaction is rolled back. This means for the exmple, it is enough to remove the entry for `acc1` from the log an execute the first two actions of transfer instead of all actions.

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

[7] We want study the two problems independently and thus assume the original implementation here.

Chapter 3

Concept

In this Chapter we will explore an Approach to handle the problem of *Unnecessary Rollbacks* described in 2.3.1. Before we can understand the solution to this problem we need to specify the technical reason of this more precisely. I was not able to find an satisfying solution for the problem of *Unnecessary Recomputations*, but it may be possible, even though the overhead is significant higher than in the first case. However, we will investigate the problem to identify and understand the challenges of this problem.

3.1 Unnecessary Rollbacks

Remember the idea given in 2.3.1. We suggested to delay the evaluation of readTVar operations to the commit phase rather than e them directly in the computation phase. While the idea would work for the example of a normal transfer, the idea would not work for the following example:

```
limitedTransfer src dst am = do
  srcBal <- readTVar src
  if srcBal < 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 does not do this. The main problem is the bind operator: `>>= :: STM a -> (a -> STM b) -> STM b`[1]. This operator allows us to extract the result of an STM action from the STM context, for example the result of a `readTVar`. This means the STM library loses any possibility to observe this value. The value is no longer in the libraries reach. Thus the library is not able to decide if the value is used to alter the control flow. Furthermore the library is not able to determine if the control flow alters when the value is modified. The only way to guarantee the ACI properties is to restart the transaction when the TVar is modified.

If the library handles a value that is **not** used for branch conditions as if it were used for branch conditions, it may loses performance, but preserves the correctness.

[1] Remember that the do notation used so far is syntactic sugar for >>= and >>

We already know an example for this. When we introduced STM in 2.1, we examined two transactions executing `transfer`. In 2.3.1 we detected that this would roll back at least one of these transactions.

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 ACI properties. Thus the way GHC chose to ensure the correctness of the implementation is to handle all values as critical values. Take for example `limitedTransfer acc1 acc2 50` and assume the initial value of `acc1` is 60. The transaction executes `read acc1` (and does not handle this TVar as a critical TVar) and determines the branch condition to be false. Then another transaction modifies withdraws `20` from `acc1`. Then initial transaction resumes and completes its computation phase. Since no TVars are critical the validation succeeds and the transaction continues. The behaviour depends on the semantics of `readTVar`. Either the TVar is read now (in the commit phase) or the TVar is not evaluated, since it was already evaluated to determine the branch condition. If the TVar is read again an then the update is commited the new value of `acc1` is `-10`. Certainly nothing you expect, when executing a `limitedTransfer`. If the TVar is not read again the new value of `acc1` is `10`. This is a lost update that generated money.

3.2 Approach

My approach to avoid unnecessary rollbacks and preserve the correctness is to handle all TVars uncritical at first. While executing the computation phase the TVars whose values are used to alter the control flow become critical. All `readTVar` operations are evaluated as late as possible, meaning a read on an uncritical TVar is executed in the commit phase and a read on a critical TVar is executed as soon as its value is used for some kind of branch, by which the TVar becomes critical.

Branch features in Haskell are the following:

- `if-then-else` expressions
- `case` expressions
- guards in functions or case expressions
- pattermatching in functions

Whenever a value is passed to one of these constructs, the TVars that the value depends on are marked critical adn the associated read operations are evaluated. A value may depend on multiple TVars. Consider the following example:

```
transaction =
  a <- readTVar t1
  b <- readTVar t2
  if a < 5
    then if b - a < 0
           then ...
```

In this example the value in the condition of the first `if` expression depends on `a`, but not on `b`. The value in the condition of the second `if` condition on the other hand depends on `a` and `b`. If `a` is greater or equal to `5`, only `t1` is critical in this transaction, otherwise `t1` and `t2` are critical in this trasaction. However the time `t1` and `t2` are critical is different because `t1` becomes critical when the first if condition

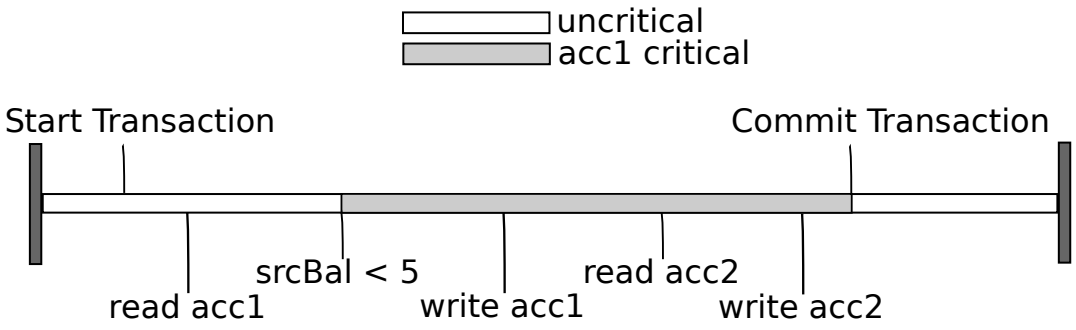

FIGURE 3.1: The critical time of the TVars in `limitedTransfer` with the alternative approach.

is evaluated and `t2` when the second if condition is evaluated. `a` is **not** evaluated again when the second if condition is evaluated. Every TVar is read at most once per transaction regarless the number of branches that depend on the TVar[2]. Even when the user evokes multiple `readTVar` operations on the same TVar, the actual TVar is accessed just once.

At the end of the computation phase all reads that are needed to decide the control flow are evaluated. All other reads that are not relevant for the control flow are not evaluated. In the `transfer` example no `readTVar` is evaluated in the computation phase, because neither of the TVars is used to decide the control flow.

Lets refrain from STM and concurrency for a second. This kind of evaluation is well known in Haskell. There are two cases where Haskell demands the evaluation of an expression[3]. The first is, if Haskell needs the value to execute an IO action such as `print`. The second is, if the value is needed to decide a branch condition.

STM is an abstraction that allows us to use sequential programming in a concurrent context. So we can use the same evaluation strategy as in the sequential context. Even if the syntax suggests it, we do not explicitly specify when the TVars is read we just specify where a value comes from and the evaluation is handled by the underlying system. Since the computation phase is processed in the STM monad, there are no IO actions allowed. This implies the only time we need to evaluate an expression is when we need to decide a branch condition. Everytime we execute other computations on TVar values and write them back or return them, this is not executed in the computation phase, because it is not needed; but increases the chances that the transaction is rolled back. By evaluating only the reads that are needed and just before they are needed we minimize the time the TVars are critical. Figure 3.1 shows the effect on `limitedTransfer acc1 acc2 5`[4]. The TVar `acc2` is not critical at any point in the transaction, because it is not needed for the control flow. `acc1` on the other hand becomes critical at the time its value is used to evaluate the branch condition `(srcBal < am)`.

When the commit phase starts some reads are evaluated and some reads are unevaluated. Before the remaining reads are evaluated the transaction locks all TVars that it has modified. The next section explains why it is enough to lock these TVars instead of all TVars that were accessed. Then the transaction is validated. This

[2]It should be clear that the actual TVars are read again, when the transaction is rolled back.

[3]There are ways other cases where Haskell demand the evaluation, but these are user defined strictness annotation suchs as `seq` or `!` for strict pattern matching or strict constructors.

[4]We assume that `srcBal` is greater than 5.

validation is similar to the validation in the original implementation; it is a comparision between the values in the log (the TVars that the transaction needed to read to determine the control flow) and the current values of the TVars. If all values match the transaction is valid, otherwise it is invalid. An invalid transaction is rolled back immediately, after the locks are released. If the transaction is valid the transaction finally executes the remaining `readTVar` operation in order to modify the actual TVars and determine return value of the transaction. Then the modifications the transaction performed are committed to actual TVar. This is the reason the `readTVar` operations can not be delayed longer. By committing the modifications the old values in the TVar are overwritten. Besides this logical issue there is a additional technical reason presented in Chapter 4. Parallel to writing the actual TVar the locks for the TVars are released. The last step of the transaction is to return the result.

Chapter 4

Implementation

We will now look upon the implementation of the aforementioned changes. Unlike to the original implementation this is not a C library, but a pure Haskell library. This brings some advantages and one disadvantages. The disadvantage is the performance as discussed in **REF TO EVALUATION**. For the costs of performance we gain a library that is easy to understand and extend. The original implementation is interwoven with the GHC runtime environment. Some STM functions are evoked by the scheduler to ensure the consistency. This makes the library sensitive to changes. To ensure the correctness of such a library is significantly harder than with a pure library, since the compiler does not aid this porcess. In other words, the devlopment of a pure library is safer and faster. (Huch and Kupke, 2005) presented a high level Haskell implementation of STM. Their aim was to provide a pure Haskell implementation that is equivalent to original implementation of (Harris, Marlow, et al., 2005). Preceding this master thesis I optimized that implementation. I replaced internally used data structure and performed two changes to the internal semantics. First, the initial high level implementation used a global lock to synchronize concurrent transactions. This coarse grained locking was substituted by an fine grained locking. Instead of a single global lock, each TVar holds its own lock and transactions acquire the minimum amaunt of locks to commit. This prevents transactions, that do not conflict, to commit simultaneously. Second, I altered the conflict detection. The initial implementation used a validation process similar to the GHC implementation. Each TVar has a queue associated. If a transaction reads this a TVar, it enters a reference to itself. If a transaction successfully commits a change to a TVar it notifies all transactions in the associated queue. If a transaction is notified it is rolled back. This way of conflict detection has the advantage that conflicts are detected earlier than before. This implementation is called the *project implementation* in the following. Figure 4.1 visualizes the development process of the libraries. We will now head over to a detailed description of the implementation developed in the course of this thesis.

4.1 STM Types

Before we head over to the implementation of the external interface of STM, we investigate the types of STM that are used in this implementation starting with the STM type itself:

```
data STM a = STM (StmState -> IO (STMResult a))
```

In its core the STM data type is a state monad. The IO type on was initially needed to perform the reads in the computation phase. As we will see later this implementation needs the IO only to create new TVars in the computation phase. `readTVar` and `writeTVar` do not need IO operations to be processed. Thus an STM action

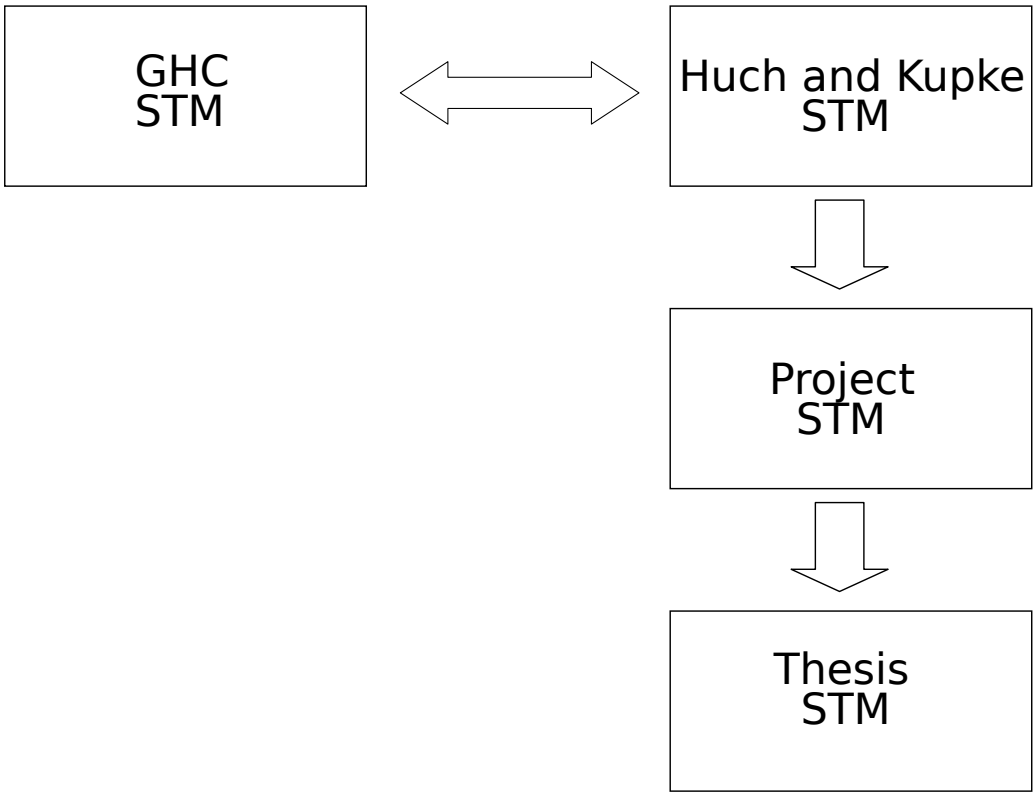

FIGURE 4.1: The STM implementations for Haskell.

takes a state and returns a result depending on this state. There are three possible results:

```
data StmResult a = Retry StmState
                 | InValid
                 | Success StmState (Maybe a)
```

The first constructor is used to indicate that a `retry` occurred. This must be distinguished from `InValid`, since `orElse` and `atomically` react differently on these results. If `Retry` is returned, `atomically` validates and rolls back if needed and `orElse` would start the second transaction. This is the reason `Retry` is accompanied by the state. The state hold the necessary information about the TVars the transaction has read. This allows to validate and possibly suspend on particular TVars. `InValid` on the otherhand is indicates always that the transaction is not valid and thus must be restarted. The last constructor is the desired outcome of an action. If the transaction does not fail, it returns `Success` and a state as well as the result. As before the state is needed for validation. The result is wrapped by the `Maybe` type. The value `Nothing` never occurs. The reason for this wrapping is explained in Section 4.2.

Before we examine the StmState, we need to take a look at the `TVar`

```
data TVar a = TVar (MVar (IORef a))
                   ID
                   (MVar [MVar ()])
                   (MVar ())
```

All TVars have a globally unique identifier called `ID`, which is immutable. The `MVar (IORef a)` hold the actual value of the TVar. The MVar is used to synchronize

multiple transaction, if they intend to access the same TVar. The `IORef` is needed to enable a correct validation, which is explained in detail in 4.2. `MVar [MVar ()]` is the queue of the MVar where transactions that wait on a change enter their their personal `MVar ()`. Remember that this is needed to delay the rollback when retry is evoked. The last part is an explicit lock for this TVar. There are currently two implementations as a result of this thesis. The details are explained later. One of these implementations lock a TVar via the explicit lock and the other by taking the MVar that hold the value.

The only thing that happens in the computation phase is that the `StmState` is computed. The core data structure is the `StmState` which holds all informations to process a transaction:

```
data StmState = TST {writeSet    :: IntMap (Maybe (),
                                            Maybe (),
                                            MVar (IORef ())),
                     notifies    :: IO (),
                     readSet     :: ReadSet,
                     retryMVar   :: MVar (),
                     uEReads     :: [Maybe ()]}
```

The `writeSet` is similar to the log in the GHC implementation. It is a `IntMap` [1]. The keys are the IDs of the associated TVars. The elements contain the `expectedValue`, the `newValue`, and `actualMVar`. The `actualMVar` is needed in the commit phase when the transaction sucessfully commits to publish its modifications. We know already the `expectedValue` and the `newValue` from 2.2, but their purpose is slighty different. The `Maybe` type indicates that the values also can be `Nothing`. This holds only for the second entry. The first entry has the `Maybe` type for the same reason the `Success` has it. The second entry on the other hand can become `Nothing`. This indicates that the TVar was read but not written by the transaction. Note that there are two kinds of *read*. The first is that `readTVar` operation is porcessed in the computation phase and the second that the current value of the TVar was read. In the original implementation these two occur always at the same time. In the new approach theses kinds of reads occur at different. For clearity I will from now on say IO-reads when I refer to a read from the actual IORef to access the current value of that TVar. Since Haskell does not provide a simple solution for inhomogeneous containers, I decided to use the type `()` and cast all values via `unsafeCoerce` before entering them to the `writeSet`. Whenever a value is taken from the `writeSet` it is casted back to its original type by `unsafeCoerce`. The ID of the TVar ensures that the type it was casted from does not differ from the type it is casted to.

`nofifies` holds an IO action that is process when the transaction successfully commits. This IO action notifies all transactions that are waiting on a TVar that is modified by the committing transaction.

`readSet` stores information about the IO-reads that where performed. `ReadSet` is defined as follows:

```
type ReadSet = IORef (InMap (IORef (),
                             MVar (IORef ()),
                             MVar [MVar ()]))
```

The need for the outermost IORef is explained later. This type is like the `writeSet` an `IntMap`. The keys are the IDs of the TVars. The entries consist of three parts. First,

[1] This refers to the IntMap in the standart libraries of Haskell: `https://hackage.haskell.org/package/containers-0.5.8.1/docs/Data-IntMap-Strict.html`

the value (in form of the IORef) that was present when the IO-read was executed. Second, the MVar the IORef that holds the value was read from. Third, the queue of the associated TVars. The exact ussage of this is explained in the next section.

`retryMVar` is a unique MVar for every transaction. This is the MVar that is entered into the queues of the TVars when needed.

`uEReads`(unevaluated read) contains all unevaluated IO-read operations. This is essential to be able to process the IO-reads that where not evaluated in the computation phase. The `Maybe` type is again for the same reasons used as it is in `StmResult`.

This concludes the overview on the data structures used and we can now head over to see how these data structures are used to implement the STM interface.

4.2 Interface Functions

Here we will inspect the implementation of every interface function.

newTVar

This is the only function (besides `atomically`) that uses IO. A TVar consists of three MVars and one ID. To ensure the uniqueness of the IDs the STM library defines `globalCount :: MVar Int` and a function `getGlobalId :: IO Int` that takes the global count increments it, writes it back and returns the old value. Due to the semantics of the MVar only one thread at a time can get an ID. No ID is assigned twice unless `globalCount` overflows. Because the chances that this happens are narrow, no countermeasures are performed to detect or avoid this problem.

All MVar are allocated by `newMVar`. This means all MVars are initialized when the TVar is created. The queue holds and emppty list the lock holds unit and the value hold an IORef with the passed value. In the end the unchanged state as well as the new TVar is returned.

readTVar

This function is the core of the new implementation. One part of `readTVar` is similar to the original Implementation. If the TVar is present in the `writeSet`, the associated value is returned. The differences are in the other part of `readTVar`. Recall the type of `readTVar :: TVar a -> STM a`. The result is a STM action. This means a function that takes a `StmState` and returns an `StmResult`. Unless the result is `InValid`, it contains a `StmState`. In other words, `readTVar` transforms the state and returns the value of the TVar. To return the value of the TVar it would be needed to IO-read the TVar. To avoid this the read is not executed directly. The function `buildVal` helps us to achieve this.

`buildVal` wraps the value with `unsafePerformIO`. The IO action consists of two parts. First, it IO-reads the current value of the TVar and returns it after wrapping in into `Just`. The wrapping with `Just` is explained when we examine the implementation of atomically. Second, entering the information to the `readSet` that are needed to validate[2]. These are the ID of the TVar as key, the MVar of the TVar that holds the value, the value that the transaction saw when it read the TVar, and the queue of that TVar. The value the transaction has seen in created by the first part of the IO action. The other information are the arguments of `buildVal`. After

[2]Notice the difference between `readSet` and `ReadSet`. `readSet` is the field of the `StmState` and `ReadSet` is the data type.

`buildVal` created this wrapping the value is (from Haskells view) like a normal expression, but the IO-read is performed when the evaluation of the expression is demanded and not earlier; simultaneously is this evaluation logged in the `readSet`. Haskell demands the evaluation of values just to decide on branch condition or to perform IO actions. IO actions in the STM monad are not allowed by the type system. Thus the evaluation in the computation phase is only demanded when the control flow depends on it.

Back to `readTVar`. `buildVal` is used to create the result. `readTVar` also modifies the the `StmState`. It adds the newly created value to `uEReads`. And it extends the writeSet. At this point there is no entry for the TVar in the writeSet yet, otherwise this part of readTVar would not be executed. A new entry is inserted. This entry contains the constructed value and the MVar that holds the value of the TVar. The `newValue` field of this entry is `Nothing`.

In the end the constructed value and the new state are returned.

writeTVar

This implementation is straight forward. Two modifications to the state are performed. One is the modifications on `notifies`. A successful commit of the transaction would mean a modification to the TVar that `writeTVar` was called on. Thus this transaction possibly needs to notify waiting transactions. An IO action is created, that notifies all transactions in the queue of the TVar. This IO action is sequenced by » with `notifies` of the initial `StmState`. The resulting action then notifies all TVars that are written by this transaction. This may lead to a action that notifies the same queue twice. This is a minor issue, becuase after the first notification the queue is emptied. To avoid this you would need to lookup the value in the writeSet, to see if it was already written and if so do not extend the `notifies` action. This look up is much likely more expensive than a notification on a empty queue (this has not been investigated).

The second modification that `writeTVar` performs is the modification of the `writeSet`. After wrapping the value for type reason with `Just`, it is entered in the first and second fiel of the associated entry in the `writeSet`. The last field of the entry is the MVar of the TVar that holds the value. There is a reason to set both, the first and second field, to the new value. If the second field is not `Nothing`, it indicates that this value is modified by the transaction and in the case the transaction sucessfully commits, the value in this field is written. The first field of the entry is the field that `readTVar` returns if it finds this entry. To avoid this doubled entry, the obvious choice were to use pattern matching to check whether the second value is `Nothing` and if not return that value. This does not work in this implementation. The problem is that the pattern matching forces the evaluation. By forcing the evaluation of an expression you may evaluate IO-reads and thus make TVars critical that are not critical. This does not happen in the current implementation, because in every entry of the `writeSet` the first field is always not `Nothing` and thus can be returned without pattern matching on this value.

In the end the new state and `()` are returned.

atomically

This is the heart of every STM implementation in Haskell; it allows the user to execute STM actions in a transactional manner. Before `atomically` starts the computation phase, it creates an initial state. This state basically holds no information and

commit with this state would result in no change at all. The the computation phase starts. In this phase atomically does not do anything but pasing the initial state to the STM action.

After the `StmResult` is calculated, atomically starts the commit phase by interpreting the result. There are three possible out comes. First, the result is `InValid`. If this occurs the computation phase is just restarted with the initial state[3].

Second, the result is `Retry newState`. This casues the transaction to lock the TVars in its `readSet` which can be accessed via `newState`. After the locks has been acquired, the `readSet` is validated. If it is not valid, the locks for the TVars are released and the transaction is rolled back like it was done for `InValid`. If the transaction is valid, the transaction enters its `retryMVar` to the queue of the TVars it has read. These queue are stored in the `readSet`. Then the transaction realeases the locks and executes a `takeMVar` on its `retryMVar`. This lets the transaction suspend until another transaction notifies it. After the transaction was notified, it removes its `retryMVar` from the queue (if that has not already happend) and restarts.

Third, the result is `Success newState result`. This is the most interessting case, because it is the commit phase and thus leads not necessarily to a rollback, in contrast to the other two results. There are currently two different implementation as a result of this thesis which only differ in this part. The performance tests presented in **REF TO EVALUATION** provide not a clear result on which implementation is better in general. I will present both implementation. When is it not clear which of the implementation we are talking about, we will refer to the first implementation as STMLA (STM lock all) and the second STMWSL (STM write set lock).

STMLA starts the commit phase by locking all TVars is has accessed. This implementation uses the explicit locks of the TVars. This information are stored in the `writeSet`, since `writeTVar` as well as `readTVar` create entries for that are not part of the `writeSet`, but processed by either of these functions. When all locks are acquired, the transaction validates its `readSet`. If this is not valid, the transaction is rolled back. If it is valid, the transaction evaluates all expressions in `uEReads`. This is done by the `seq` function, which forces the evaluation of its first argument and returns its second argument. As always in Haskell the expression is evaluated to WHNF[4]. This is where the `Maybe` type comes in handy. `buildVal` created a `Maybe` value that was wrapped with an `unsafePerformIO` action. If we force the evaluation of this expression, the IO-read is performed, but the underlying expression is not evaluated, because it is wrapped in the `Just` constructor. Without this constructor, we would evaluate the expression that is stored in the TVar. Imagine this expression is a complex computation. This would mean, we would suffer a performance overhead in the case, that this computation is not needed. Even worse we execute this in the commit phase, while we hold the locks for many TVars. To maximize parallelism, we aim to minimize the time in the commit phase. Besides these performance problems, there is a serious semantic problem. Everytime we force the evaluation of an expression that is not needed, we change the semantics of the program. This may lead to exceptions or loops that would not occur normaly. After the `uEReads` are evaluated, the actual writes are prepared. The writes are divided in two parts. First take the value from the MVar and second write the new value. Preparing means all the value holding MVars of all TVars that are going to be written are emptied. Then the `notifies` are processed. At last the new values are written

[3] Actually the `readSet` needs to be discarded explicitly, because it is an IORef and thus would not be empty by taking the initial state.

[4] In case you do not know what *weak head normal form* is please refer to `https://wiki.haskell.org/Weak_head_normal_form`

to the value holding MVars. This is needed to prevent a very nasty bug [5]. The last step of the commit phase before returning the result is to realease the locks that were taken at the start of the commit phase.

The second implementation (STMWSL) starts the commit phase by validating. As before, the transaction is rolled back if it is invalid. If it is valid, the remaining reads in `uERead` are evaluated. After all reads have been processed, the actual writes are prepared. This means the TVars that are modified by the transaction are accessed and their value holding MVar is emptied. At this point no other transaction is able to either read or write these TVars. In the first implementation the explicit locks were taken, which does not prevent other transactions from reading these TVars. When the writes are prepared, the transaction is validated again. In contrast to the first validation, this validation is mandatory. It is possible that the current values of the TVars have changed after the reads were processed and before the writes were prepared (and by this the TVars locked). The first validation is added to avoids the problem that the transaction acquires the locks, although it is invalid. The costs for locking are much higher than the costs for validation. If the transaction is invalid after the reads are processed and the transcation acquired the locks, it is rolled back. If it is valid the it is similar to the first implementation (STMLA). At first the `notifies` are processed and then the writes are processed. At last the result is returned. Note that it is not necessary to unlock the TVars explicitly, because only the value holding MVars of the TVars serve as a lock in STMWSL. By filling these MVars the associated TVar is unlocked.

Both implementation have their own benefits. STMLA does never roll back, when the transaction does not branch at all. If two transactions read the same TVar they cannot commit at the same time in STMLA. In STMWSL it is possible that two transaction, that read the same TVar, commit simultaneously (if non of them also writes this TVar). It is also possible that a rollback occurs, even if the transaction does not branch. The problem is that the remaining reads are evaluated in an unlocked context. This means that other transactions can modify these values after they were read and thus invalidate the reading transaction. It would be better, if the transaction process the `uEReads` after it has locked the TVars. This is currently not possible. As we will discuss in detail in the next section, when we only lock the modified TVars, it is essential that no other transaction reads them while we hold the locks. Otherwise the ACI properties cannot be guaranteed. If we take the value holding MVars it prevents other transactions from reading the associated TVars. Unfortunately, this also prevents us from reading the TVars. That is why we need to process `uEReads` before we acquire the locks for the TVars. The clever reader may have noticed that we get the values of the TVars when we prepare the writes, because we execute `takeMVar` on the value holding MVars. To use these information would require the `unsafePerformIO` actions produced by `buildVal` to behave diffrently when it is executed in the computation phase and in the commit phase. Nevertheless, this topic remains for future work and is discussed in **REF TO FUTURE WORK**.

A solution that combines STMLA and STMWSL would be better[6]. A combined solution locks the TVars that the transaction has modified in a manner that not **other** transaction can read them. The transaction itself is still able to read the locked TVars. Then the transaction is validated. If it is valid, the remaining reads are processed and

[5] For interessted readers is this bug explained in the appendix.

[6] This could not be proved, because no such implementation is present currently.

the transaction publishes its modifications. At last the transaction returns the result, after unlocking the TVars.

»=

The monadic bind operator, »=, is an important part of the interface, because it allow us to use the do-natation. The aim of this thesis is to provide an alternative STM implementation for Haskell without altering the semantics or interface. One of the most comfortable things about is the way we use it. Thus it is important to retain this feature. `»= :: STM a -> (a -> STM b) -> STM b` allows us to extract the result of an `STM` action from the STM context and use it to create a new STM action. Additionally it is used to translate `<-` of the do-notation. The implementation is straight forward, but for the sake of completeness presented nonetheless. The result is a STM action, meaning that it is a function that takes a `StmState` and its result is a `StmResult`. Thus the function basically has three arguments. The `STM` action, the function of type `a -> STM b`, and the `StmState`. These arguments are called *passed action/function/state* in the following.

The resulting function first applies the passed state to the passed action. If the result is `InValid`, it just returns `InValid`. If it is `Retry newState` it returns `Retry newState`.

Note that it is important to pass the `newState` created by the action, because it contains information that are needed to wait before rolling back.

If the result of the action is `Success newState res`, the function first needs to unwrap `res`; it is wrapped with the `Just` constructor. This is done by the partial function `fromJust`. The implementation ensure that `Nothing` can never occur and thus the call to `fromJust` is safe. After the value is extracted, it is applied to the passed function. This application results in a `STM` action. To gain `StmResult` as the result of the function the `newState` is applied to the action. Thanks to the laziness of Haskell the call to `fromJust` is not evaluated immediately. This is the key of this implementation, since we avoid the evaluation of the IO-read by this. If `fromJust` would be evaluated when the bind operator is evaluated, a action like `a <- readTVar t1` would not differ from the original implementation, because the `unsafePerformIO` action would be executed immediately. Since we use a `case` expression to branch on the different `StmResult`s, we could also extract the value from the `Maybe` type via pattern matching instead of `fromJust`. This would also lead to the evaluation of the value and make the wrapping performed by `buildVal` futile.

retry and orElse

The implementation of `retry` and `orElse` are very simple. `retry` is a STM action. The passed state is wrapped with the `Retry` constructor. That is all `retry` does.

`orElse` executes the first acion with the passed state and if the result is not `Retry newState` it just returns this result. If it is `Retry newState`, the second action is executed with the passed state (not the `newState`) to discard the writes executed by the first action. Since the `readSet` is a `IORef`, we do not lose the information on the TVars that was read.

4.3 Notes on the Implementation

Some details of the implementation remained unexplained until here. These details are highlighted in this section.

Deadlock Avoidance

One Motivation for STM is the deadlock freedom. We have seen in Section 2.1 that acquiring multiple locks always includes the danger of deadlocks if multiple threads work concurrently. There are two concepts to avoid deadlocks for such settings. The setting is that multiple threads try to acquire and arbitrary number of shared locks. One avoidance strategy is to realease all locks and try again if we try to acquire a lock that is already locked. The other avoidance strategy is to acquire the locks in a global order. STMLA as well as STMWSL use the latter. The TVars that need to be locked are stored within `IntMap`s. If we try to lock these TVars, we convert the `IntMap` to a list and process it. Since the conversion from an `IntMap` constructs an ordered list, we lock always in a global order. The TVars are ordered by their IDs.

4.3.1 MOVE THIS TO THE IMPLEMENTATION CHAPTER

When the commit phase begins the transaction is validated. The validation does not differ from the validation in the original implementation. When a `readTVar` is evaluated the current value of the TVar is logged. Validating the transactions means to check if the logged values match the current values of the TVars. If the transaction is not valid, it is rolled back. If the transaction is valid, the TVars that the transaction want to modify are locked and the transactions is validated again. This is needed, because it is possible that a TVar in the log was modified after the validation and before the lock was acquired. This seems to be an unnecessary overhead, but has a slightly better performance than first locking and then validating. This is discussed in Chapter**REF TO EVALUATION**. The original implementation also uses this scheme, which was proposed by Fraser (Fraser, 2004, Page 42). The reason is the high cost for locking the TVars. This locking itself is not particular expensive, but it hinders parallelism. Everytime a TVar is locked no other transaction is able to read, lock, or validate this TVar. Additionally if a TVar tries to access a locked TVar, it is suspended and a context switch follows. Context switches in Haskell are not as expensive as context switches of OS threads, but it schould not be neglected, especially when dealing with a large number of threads. Validation is a operation, which needs two memory access per entry in the log. One memory access is the access to the log entry to look up the expected value and the other memory access is the access to the actual TVar[7]. This is reasonable considering that the log only consists of TVars that are needed to determine the control flow. So validation is significant faster than locking.

At this point no other transaction is able to modify the TVars and thus the evaluation is safe in the sense that the value cannot change until the commit of the transaction is completed. After the reads are evaluated the writes are processed and the result is returned after unlocking the TVars.

[7]The log entry needs to be accessed two times. First to get the expected value and second to get the next entry, because the log is a dynamic structure and thus needs a pointer to the next entry comparable to a list. Nevertheless the entry itself is (most likely) in one block in the memory. This is why I consider one memory access to be enough for the entry.

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:

Fraser, Keir (2004). *Practical lock-freedom*. Tech. rep. University of Cambridge, Computer Laboratory.

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.

Huch, Frank and Frank Kupke (2005). "A high-level implementation of composable memory transactions in concurrent haskell". In: *Proceedings of the 17th international conference on Implementation and Application of Functional Languages*, pp. 124–141.

Larus, James R and Ravi Rajwar (2007). *Transactional memory*. Vol. 1. 1. Morgan & Claypool Publishers.