CHRISTIAN-ALBRECHTS-UNIVERSITY

MASTER THESIS

Alternative Software Transaction Implementation in Haskell

Author:
Lasse Folger

Supervisor:
Dr. Frank Huch

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

in the

Programming Languages and Compiler Construction
Department of Computer Science

January 10, 2017

Declaration of Authorship

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

Kiel, January 10, 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 5
2.2.1 Computation Phase 5
2.2.2 Commit Phase 7
2.2.3 Notes on the Description 8
2.3 Problems 8
2.3.1 Unnecessary Rollback 8
2.3.2 Unnecessary Recomputations 10

3 Concept **11**
3.1 Unnecessary Rollbacks 11

A Frequently Asked Questions **13**
A.1 How do I change the colors of links? 13

Bibliography **15**

List of Figures

List of Tables

List of Abbreviations

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

Chapter 1

Motivation

Modern computer architecture includes multicore processors. To utilize these multicore system to their full extend, concurrent and parallel programming is needed. By this new challenges arise. One challenge is the logical issue of splitting the problem in smaller problems which can be processed by different threads in parallel. Aditionally there are technical challenges. 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 problem when used in the parallel threads. The problem is that these operations are non atomic operations. Thus (`x = x + 1`) consist of three parts. first reading the old value, second adding `1`, and thrid write the new value. This means two threads in parallel can both read the old value, then both add `1` to the old value, and then write 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.

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

This seem to be a simple construct, but its capabilities are enormous. It is highly complex to use a semaphore correctly. The main problem of semaphores is the so called deadlock[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 the so called software transactions were introduced (Harris et al., 2005). Sofware transactions are inspired by the long known database transactions (Gray

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

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

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

Chapter 2

Introduction

2.1 Software Transactional Memory

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

```haskell
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:

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

Thread 2:

```haskell
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:

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

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

[2]In fact is `transfer acc1 acc1 50` enough to evoke a deadlock

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

We can use STM to avoid these problems. STM provides a single element buffer named `TVar`. A TVar always holds an value and is never empty. TVars are read and written with the functions `readTVar` and `writeTVar`, respectively. In contrast to `putMVar` and `takeMVar`, the TVar operations are not `IO` actions but `STM` action[3]. `STM` is an instance of Monad, hence multiple STM actions can be combined using the comfortable do-notation. The following code represents the example from above implemented with TVars rather than MVars:

```
type Account = TVar Int

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

Note the type of transfer is no longer an `IO` action, but an `STM` action. Besided 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 exmaple evokes no deadlock:

Thread 1:

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

Thread 2:

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

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

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

These properties explain the name `atomically`, because the enclosed code appears to be executed instantaneously wihtout any interactions with other threads. Before we turn over to the implementation of STM, we take a deeper look at the interace of the STM.

[3]For those who are wondering when I use `SMT` and when STM. I use `STM` when I refer to the Haskell type contstructor 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 frist transaction fails (retries), the second transaction is executed instead of the first one.

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

2.2 Implementation

I will now give an overview on the current implementation of STM in Haskell. For a detailed description of the implementation refer to (Harris et al., 2005) Afterwards I will analyize the problems of this implementation.

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

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

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

Take a look at the following example:

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

This code would lead to the following read and writeSet:

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

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

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

`retry` aborts the computation and returns a 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 original 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 wirteSet is copied (name it ws1 for now). Afterwards `trans2` is executed. If the execution leads to `retry`, the writeSet is set to ws1 and `trans3` is executed [4]. The readSet remains unchanged, thus the TVars added by `trans2` are included. If the execution of `trans2` leads not to retry, the result, the readSet, and the writeSet produced by `trans2` are the result of `transaction`. In that case `trans3` is ignored.

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

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

- `writeTVar`: Add TVar and value to writeSet
- `readTVar`: Lookup the value in writeSet, if not present read the original TVar and add TVar and version number to the readSet.
- `newTVar`: Create and initialize a new TVar.

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

- `retry`: Return a result that indicates a failure.
- `orElse`: Backup the writeSet, execute the first transaction, if it fails restore the writeSet and execute the second transaction, else return the results of the first transaction.

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. There are two possible results. First the result of the computation phase indicated a failure (i.e. engaged a `retry`). When this happens the transactions first validates (the next paragraph explains this in detail). 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.

If the computation phase return a result that indicates a success, the transaction also validates. Remember the readSet is a collection of pairs matching a TVar to a version number. To validate this it is sufficient to compare the version numbers stored in the readSet with the version number in the actual TVars. If all version numbers match it means the transactions has seen an consistent state of the system, because no TVar hast 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 has changed after the transaction read it. Thus the isolation is violated and the transction is invalid. At this point all TVars in the readSet are locked to avoid race concurrency problems.

Depending on the result of the validation the transaction is further processed. If the the transaction is valid, aditionally the TVars in the writeSet are locked (the TVars in the read set 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 transction to the caller of `atomically`.

If the transaction is found to be invalid, the transactions is rolled back after the locks for 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 on 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. Since `transcation1` is valid, it commits its modifcations, which means 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.

Since t1 is unlocked transaction2 can validate. Since the version number of t1 now is 1, but the version number in transaction2s readSet is 0. This means the validation fails and the transactions is rolled back. If there were no validation, transaction2 would also commit at this point. Thus the value of t1 is set to 6. This is obviously the well known concurrency problem named *lost update*.

2.2.3 Notes on the Description

In the actual implementation the validation is not only peformed at the end of the transaction, but everytime the transaction is dispatched by the scheduler (is that still true????? it was in the original work, but due to the multiprocessor work around it may is outdated allready)

2.3 Problems

I will now explain two problems with this implmentation. These problems can be examined independently.

2.3.1 Unnecessary Rollback

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

Thread 1:

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

Thread 2:

```
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:

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)
```

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 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 parallel. This means both have read the initial values of acc1 and acc2 and added these information to their readSet. Furthermore have both transactions added entries for `acc1` and `acc2` to their writeSet. Now both transactions try to commit, thus try to lock the TVars. If `Thread 1` gets the locks for `acc1` and `acc2` it validates and commits, since none of the TVars has changed after `Thread 1` read it. After that `Thread 2` aquires the locks and validates. Since `Thread 1` changes the version numbers of `acc1` and `acc2` by committing, the validation fails and the transactions of `Thread 2` is restarted. In conclusion no performance improvement was achieved. Both threads are still executed on after another and not parallel as intended.

Let us rearrange the operations of `transfer` to see how this can be improved. Note that we can rearrange the operations without changing the semantics of the resulting code due to the ACID 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 accound 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 could 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 change the semantics of `readTVar` by delaying the evaluation and nothing else, 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 pretty common. Haskells evaluation strategy is (NOCHMAL GENAU WERDEN)*lazy*(NOCHMAL GENAU WERDEN). That means passing values that are unevaluated is normal.

After the computation phase the commit phase follows. The first step is to lock the accessed TVar in order to perform the validation. Since both transactions used the same TVars, they will commit successively instead of parallel. Let us assume `transaction1` gets the locks first and trys to validate[5]. Since the readSet is empty the validation unnecessary; it always succeeds. 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

[5]You could argue that `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 threads at that point.

readSet and the actual value is return. Since the readSet is allready this step is futile, but for now will be performed.

After `transaction1` finished and released the locks, `transaction2` aquires these locks and validate. The readSet of `transaction2` is also empty, thus the validation succeeds. Now `transaction2` commits its writeSet and completes by releasing the locks.

Both transactions run parallel as far as possible and did not roll back. In 3 we will see the challenges that arise, if we want to implement this idea and how to engage these challenges.

2.3.2 Unnecessary Recomputations

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

```
transaction = do
  a <- readTVar t1
  writeTVar t1 (f a)
  b <- readTVar t2
  writeTVar t2 (g b)
```

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 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 transcation is rolled back. Which means the readSet and writeSet are discarded, regardless which TVar was the reason for the failed validation.

For example if `t1` was modified after the transaction read it, the transaction is rolled and both statements are executed agains. This includes the read and write of `t2` although the value of `t2` did not change. Hence the exact same code with same inputs and the same (relevant) environment is executed twice. If we were able to invalidate parts of transactions instead of transactions as a whole, we can save time when rolling back a transaction.

Some may argue it is not common to execute independent statements in a single transaction. But I do not want the performance of the STM library to rely on the capabilities of the programmer. Furthermore do I not want to make any asssumption on how the library is most likely be used. I am not sure if there are cases in which the execution of independent statements is needed, but I do not want the user to think about this issue. The user should focus on the correctness of the code; the performance is the task of the library.

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: »= `:: STM a -> (a -> STM b) -> 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 it 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 loses performance, but preserves the correct the 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.

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

Appendix A

Frequently Asked Questions

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

The color of links can be changed to your liking using:
`\hypersetup{urlcolor=red}`, or
`\hypersetup{citecolor=green}`, or
`\hypersetup{allcolor=blue}`.
If you want to completely hide the links, you can use:
`\hypersetup{allcolors=.}`, or even better:
`\hypersetup{hidelinks}`.
If you want to have obvious links in the PDF but not the printed text, use:
`\hypersetup{colorlinks=false}`.

Bibliography

Dijkstra, Edsger Wybe (1965). “Cooperating Sequential Processes, Technical Report EWD-123”. In:

Gray, Jim and Andreas Reuter (1992). *Transaction processing: concepts and techniques*. Elsevier.

Harris, Tim et al. (2005). “Composable Memory Transactions”. In: *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP ’05. Chicago, IL, USA: ACM, pp. 48–60. ISBN: 1-59593-080-9. DOI: 10.1145/1065944.1065952. URL: http://doi.acm.org/10.1145/1065944.1065952.