1 27.10.2016

First implementation of the new STM design was finished. It allows multiple transactions to commit without aborting each other. The transaction only aborts if a constraint is violated.

1.1 TVar changes

The first part was to extend the TVar by a constraint Field, thus it looks like this now:

```
data TVar a = TVar (MVar a)
                   ID
                   MVar [MVar ()]
                   (a -> Bool)
                   (MVar ())
```

At this point the constraint is a static funktion, which must be specified at the point the TVar is created. For later implementation this may be modified, so the constraint can be exchanged later on.

Furthermore was the waitqueue kept even if there is no longer a direct *retry*. When a transactions tries to commit its changes, it validates first to make sure no constraint is violated after its commit. If a change would violate a constraint the transaction enters its retryMVar into the waitqueue of the TVars that it read and suspends until another transaction changes a TVar that was read.

An explicit lock were introduced. If the *MVar a* would be used for locking, its value has to be remembered and looked up everytime this TVars values is used.

1.2 STM interface

STM is no longer a Monad. This means the user is no longer able to use the results of STM actions directly as values. For example he cannot use the result of readTVar as a value to evaluate a branch condition or pass it to writeTVar. That is the reason the type of writeTVar had to be changed to:

```
writeTVar :: TVar a -> STM a -> STM ()
```

writeTVar now takes a STM action instead of a value. Internallly the result of the STM action is extracted and written into the writeSet. This is necessary for later readTVar operations.

In order to allow the user to change values after they were read, the function *modifyVal* were established:

```
modifyVal :: STM a -> (a -> a) -> STM a
```

The resulting action behaves like the passed action but its result is modified by the passed function. An example use is the following function, which reads a value and modfies it:

```
readAndModify :: TVar a -> (a -> a) -> STM a
readAndModify tv f = res
  where val = readTVar tv
        res = modifyVal val f
```

1.3 Sequence

A new type class was introduced to be able to compose multiple STM actions, but prevent the direct use of values.

```
class Sequence f where
  (<%>)  :: f a -> f b -> f b
  result :: [f a] -> f [a]
  infixl 1 <%>
```

The $< \% >$ operator is similar to the Monad operator $>>$; thus the user is able to sequence multiple actions without being able to exctract the value. Later on I may define some laws that this operator must fulfill.

The result funktion has a functionality similar to *sequence*, which is defined on Monads. It takes a list of actions. The actions are combined to a single action whose result a list of the results of the inner actions is. This function is needed to be able to read or create multiple TVars in a single transaction.

Both of these functions are used to pass the StmState from one action to the next.

Later on I may implement a function *lift* to be able to create STM action whose result is a specific value. This is usefull to be able to enter values into TVars whith a call like this:

```
writeTVar t1 (lift 42)
```

At this point I am not sure if this is needed at all, but it will be considered.

1.4 Performance

To find out if the changes made in the STM implementation are usefull, we performed the first timing tests. The test were promising. We used the tests that were made for the implementation that was developed in the project. The here developed version were faster than the base version and the version with direct notification.

2 todo

Further researches are needed to determine if the current implementation can be combined with the monadic version, to allow specific transactions to branch dependent on the content of the TVars.

Dynamic constraints: find out if it is possible to let transactions change the constraints on TVars.

No explicit lock: try to use the MVar a which stores the value in the TVar could be used for locking in order to archieve true consistency.

Think of a way to detect which TVars led to a retry and only wake up if one of those is changed.

Implement direct retry. Maybe by extending the STMResult again by a Retry constructor

Laws for Sequence

Find out if *lift* is needed or makes any kind problem.

Rewrite the other test cases for STMC and test them.

3 28.10.2016

Worked on the todos from 27.10.2016

3.1 lift

We implememted *lift*. We decided to implement lift, because we implemented an instance for Applicative. We had no possibility to combine the values of two TVar. With Applicative we obtained two usefull operations. The first was $< * >$ which now replaced *modifyVal*. Wih modifyValue we were not able to combine the results of two different TVars with an arbitrary function. To be able to use arbitrary functions the function *pure* in combination with $< * >$ are used. Take a look at the following example:

```
readAndModify :: TVar a -> (a -> a) -> STM a
readAndModify tv f = pure f <*> readTVar tv
```

We allready know this function from 1.2 where it was implemented with modifyVal. This version uses the functions provided by Applicative.

3.2 Constraints

We implemented the function *replaceConstraint* with the following type:

```
replaceConstraint :: TVar a -> (a -> Bool) -> STM ()
```

This function allows the user to change the existing constraint on a TVar. In order to allow TVar values which do not violate their own constraint the commit phase was split in two parts. In the first part the new constraints are checked. In the second part the existing constraint that were not changed are checked (if they were written). If any constraint is violated, the transaction retries.

3.3 testcases

The other testcases could not be reimplemented in the STMC interface, because they branch dependent on the values of the TVars. New testcases will be produced later.

4 01.11.2016

We will use the following notations to distinguish the implementations of STM. The first is STMM. STMM is the monadic implementation of STM. The monadic version is the one GHC uses and that allows the user to branch on values. The second is STMC. The C stands for the constraints which are used to roll back transactions. The version that is developed here.

We combined the two implementations STMM and STMC. This means we are able to work with STMM and STMC transactions on the same values. The STMM transactions are aborted as usual, when a value that was read is modified or a constraint is violated. The STMC transactions aborted when a constraint is violated, but **not** when a value that was read is modified.

The combination was tested briefly. We introduced a new test case in order to test their interactions. In this test a (predefined) number of TVars is created. For each TVar a thread is created. This thread increments the TVar with a STMC transactions in a loop. Meanwhile executes the main thread an STMM transaction. This transaction reads all TVars sorts their values and writes them back. Afterwards the values are printed. This behaviour is also looped.

The test results were not satisfying, because the STMM transaction hardly ever finshed even for a small number of threads. This transaction needs access to all TVars. If one TVar is modified during the transaction, it is aborted.

The basic concurrent test were adapted to the combinational use. Half of the transactions now use STMM transactions half of them STMC transactions. In addition is the waiting of the main thread implemented with STMC.

The test result were satisfying. The result were the expected one which means no lost update or dirty read were performed. Furthermore were the STMC and STMM transactions used concurrently.

5 02.11.2016

A new concept named *LazyValue* was introduced, implemented and tested.

The goal of LazyValues is to combine STMM and STMC in one library and to make it possible to use both opereations in a single transaction. This was archieved by extending the STM interface by the following functions:

```
readTVarLazy  :: TVar a -> STM (LazyValue a)
writeTVarLazy :: TVar a -> LazyValue a -> STM ()
Instance Applicative LazyValue where
evalLazyValue :: LazyValue a -> IO a
```

readTVarLazy is similar to a read in STMC. The value cannot be accessed directly. This means no branching on this value is possible. Due to the instance of Applicative this value can be modified or combined with other LazyValues. *writeTVarLazy* allow the user to store LazyValues in TVars. The normal interface of STM is preserved. At last the function *evalLazyValue* is needed. This function allows the user to convert LazyValues to normal Values. It is needed if the result of a transaction is a LazyValue.

If a LazyValue was written and is then read by *readTVar* (not lazy) the transactions enters its retryMVar into the waitqueues of the TVars which contributed something to this value. For example:

```
a1 <- readTVarLazy t1
a2 <- readTVarLazy t2
writeTVarLazy t3 (pure (+) <*> a1 <*> a2)
readTVar t3
...
```

If transaction1 executes this code and another transaction modifies *t1* or *t2* before transaction1 finishes transactions1 is rolled back, although it performed a lazy read on *t1* and *t2*. Since transaction1 performed a strict read on *t3*, it is allowed to branch on this value. Additionally depends *t3* on the value of *t1* and *t2*, thus changing these values may change *t3*.

6 Alternative

We also investigated Alternative and its possibilities for STM. The functions of Alternative are already defined. $< | >$ is known as *orElse*. Further more is *empty* defined in the form of *retry*. Since STMC has neither retry nor orElse, no Alternative definition for STMC was made. For the monadic version an Alternative instance where defined. This allows to use the functions *some* and *many*.

some and many are of similar type ($Alternative f \Rightarrow fa \rightarrow f[a]$). Both functions execute the passed transaction repeatedly until it fails. There is one difference between these functions. Some executes the transaction at least once, while many returns the empty list, if the transactions runs into an retry in its first execution. If these functions are usefull has yet to be seen.

7 Arrows

Another structure which generalizes Monads are Arrows. The core idea of Arrows is to combine programm fragments. The core idea of arrows is, that an computation is something that takes an input and converts it to an output. It is a highly functional construct and not meant to be used in concurrent context. That is the reason arrows seem not to be usefull for transactions. Arrows allow no kind of sequencing unrelated actions. An operation like $>>$ or $*>$ ist missing. This makes the Arrow class uninteresting for STM. Furthermore is no if-then-else like contruct given which makes this class worse than Applicative with regards to the usability.

8 Graph

We discussed the possibility of using a dependency graph on TVars to avoid unnecessary recomputations. To be exact in the case of a roll back we could look up in the dependency graph which TVars are affected by the invalidation of (a) specific TVar(s). This graph can be constructed, but to construct it in a way that it holds the information, which are needed to do a reevaluation, is costly.

We implemented the dependency graph based STM library. With this implementation only part of the TVars are reevaluated, if the transaction rolls back. The reevaluation is done in the commit phase while the TVars are locked.

For every STM action within the transaction a node in the graph is added. Another approach would be to add for every TVar a node in the graph, but this could lead to unnecessary reevaluations. The Nodes consist of three elements. First a MVar () that is needed to be able to invalidate the node. The second is a (Maybe (IO a)). This element hold the IO actions whose result is the value that should be written in the MVar. The third element is the ID of the TVar the STM action is working on. The id is need for the reevaluation to map the nodes to their associated TVars.

Since there is no information about the STM actions which led to this graph, the reevaluation can only be done once. To make sure it does not interfere with other commits the reevaluation is done, while the TVars are locked. In other words the implementation works like this:

```
transaction
lock
validate
  if valid
    then commit
         undlock
         return res
    else reevaluate
         unlock
         return res
```

We noticed that this implmentaion is not usefull at all, because the evaluation is performed in the time the transaction locked the TVars. To be exact, we sequentialized the transactions.

9 LVars

We read the paper about LVars[1]. These LVars aim to provide deterministic prallelism, but they are restrictive to the use of the LVars. The operations used on the LVar values need to be commutative. Furthermore needs the value of the LVar to be monotinic. This is too restrictive for the use in the transactional context.

For example consider Intergers. If we apply the restrictions to this data type, it meand we may only use addition and multiplication, but not subtraction. In the case of Booleans we need to define first which means monotonicity. With this restriction we cannot even implement the most basic examples for concurrent programming. The dining philosopher or a simple bank account.

Furthermore is the main objective of LVar the deterministic parallelism and not the preformance. Since the main objective of this thesis is to optimize the existing implementation of STM performance is the main objective of this thesis, while handling nondeterminism by detection and correction.

10 Papers

The paper *Applicative* summarizes the motivation and implementation of the typeclass Applicative. The Applicative class is compared to Monad and Arrows. It is the Paper the Haskell implementation of Applicative is based on.

In *Composable Memory Transaction* an alternative implementation for *stm* is givin, which is a pure Haskell implementation rather than C library, which is linked at compile time. This allows a better development and maintaining of the library.

doublyLinkedLists.pdf presents a lock free synchronized doubly linked list which is based on CAS. Furthermore is a lock free deque presented.

guranteeing determinism describes a concept for parallel programming in Haskell by explicitly declaring the evaluation strategies. Therefore the operators *par* and *pseq* were established.

hybrid presents an approach to use the efficiency of hardware transaction to improve the performance of software transactions which are not as restricitve as hardware transactions.

i-structures presents which are used to gain parallelism without losing determinism. To do this the values are saved in i-structures which can be defined at any time and hold an (not yet) computed result. If a someone tries to read on this structure the reader is blocked until the computation is finished.

lockfreeData introduces the research field of lock free data structures. At first the primitives are presented and afterwards data structures are presented. A lot of references how and with which primitives the data structures are implemented is givin.

LVars presents an appraoch to archieve deterministic parallelism by restricting the data structures to lattice based monotonically growing data structures. This is a theoretically and formal work rather than an pratical work with a concret implementation.

optimising STM is a paper which optimizes software transactions by engaging technical implementation issues such as the shadow copies and lookups in the write set. This appraoch is based on a pessimistic implementation and specific ussage assumptions.

stmDatastructures is a paper written by the *stm*-Authors. In this paper they compare the performance of synchronized data structures based on locks and synchronized data structures based on transactions. Transactions performed better.

unread introduces the function *unread* which removes a TVar from readset in order to allow to use TVars for traversable data strutures. This is important for different transactions, which are working on different parts of the list, to prevent them from invalidating each other.

SetTransaction describes an implementation of transaction based Sets by integrating the stm implementation into the set implementation. This is a specialized use case and the authors do noch try to generalize this idea for arbitrary data structures.

ApplicativeDO describes the Motivation and implementation behind applicativeDO. Sadly this is not appliable to my STM Implememtation. Applicative do is too restrictive. Everytime an expression has a dependency to a aforebinded variable, bind is used. My implememtation just wants to use bind if there is an branch condition dependency for the binded value.

CaPRA stands for Automatic Checkpointing and Partial Rollback (no clue about the letter ordering). It described ar concept of creating checkpoint while a transcation is executing. In a case of a conflict the transaction is not completely rolled back just to a suitable checkpoint.

[2] is a book about parallel programming and transactional memory in general, not specific to Haskell.

book is a book about Transactional Memory in general. The first part of the Book describes the challenges and design design decision for Transactional Memorys. The second part gives a broad overview on existing TM systems. Especially the definitions for Transactional Memory such as optimistic and pessimistic may be worth citing.

stm-invariants is a paper about extending STM by invariants. The paper provides an (extended) operational semantics for the haskell STM and explains the implememtation in the current system. This can be used if the new system should support invariants aswell.

11 Definitions

A Collection of the Definitions I will use for the course of this thesis.

11.1 Critical Value

A Value is considered *critical* for transaction T, if its modification results in a rollback of T. Up until now this are values which were bind by the transaction at some point. Technically this is only needed if the value influences the control flow. In other words if the evaluation was demanded.

11.2 Chunk

A *chunk* is a part of a transaction which is inseparable. Seperators are $>>$ and $*>$, since these operations are resetting the dependencies. Chunks are always executed as an unit and can therefore not be separated.

12 (Fixed) Bugs

12.1 2 Phase Commit

08.12.2016. While testing some transformations rules I engaged a bug, my tests have not detected until now. Up until now the commit phase iterated once on all TVars. For each TVar the associated action were executed and **immediately** written. Take a look at the following example to understand were this fails:

```
trans = do
  readTVar t3 **> writeTVar t1
  readTVar t2 **> writeTVar t3
```

You expect that after this transaction the initial value of t2 is in t2 and t3 while the inital value of t3 is in t1. This was not the case since first the write for t3 were processed. This means the value of t2 is read and written (on IO level) to t3. Then the write for t1 is processed. This means reading t3. Since t3 was already modified on IO level, only the new value is available. The result is that all three TVars contain the same value.

FIX. The fix of this bug was simple. The commit phase was devided into two sub phases. First **all** actions are evaluated and after that writing on IO level is processed. By this always the initial values are read. This does not mean that examples like:

```
trans = do
  writeTVar t1 (pure 42)
  a <- readTVar t1
  if a == 42
    then return ()
    else loop
```

won't work any longer. This is handled by the write set of the STMState. Since the IO level action for reading a TVar is only created if the TVar is not allready in the write set.

12.2 Exception handling

09.12.2016. The implementation cannot handle exceptions.

12.3 Unshared reads

05.01.2017. When readTVar is executed, an IO action is created, which is evaluated in the commitphase. If the result of readTVar is distributed to several points in the program, this IO action is evaluated multiple times. To avoid this, we need to be able to share the result of IO actions while retaining the control of execution time. We cannot share the result of an IO action before it was executed. But we do not want to execute the IO action before we share it. Sharing is done in the calculation phase, while execution is done in the commit phase.

A solution that may would have wokred would be the explicit sharing library, which is no longer supported by the ghc 8.0.

19.1.2017 The problem was fixed by wrapping the values into *unsafePerformIO* actions. These are either evaluated when the value is bind to some variable or in the commit phase. The values return by *readTVar* are of the following form:

```
unsafePerformIO $ readIORef ref >>= return . (,) deps
```

ref is the IORef of the associated TVar and *deps* is the waitqueue of that TVar.

References

[1] Lindsey Kuper and Ryan R Newton. Lvars: lattice-based data structures for deterministic parallelism. In *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing*, pages 71–84. ACM, 2013.

[2] James R Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2007.