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*.