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)
```

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 it read and suspends until another transaction changes a TVar that was read.

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 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.