Composable Memory Transactions in Concurrent Haskell

Frank Huch and Frank Kupke

University of Kiel, Institute of Computer Science
Olshausenstr. 40, 24098 Kiel, Germany
{fhu,frk}@informatik.uni-kiel.de

Abstract. Composable memory transactions are a new communication abstraction for Concurrent Haskell, which provides the programmer with a composable communication concept. Unfortunately, composable memory transactions are implemented as external functions for ghc version 6.4 and not available for other implementation of Concurrent Haskell. We present an implementation of memory transactions within Concurrent Haskell. The presented library can be executed within older ghc versions as well as with the popular Hugs system. Benchmarks show, that our library performes well. Furthermore, our (high-level) implementation can be extended and maintained more easily than the low-level implementation provided by ghc 6.4.

1 Introduction

Harris, Marlow, Peyton Jones and Herlihy proposed a new communication abstraction for Concurrent Haskell [10, 9], called *software transactional memory* (STM) [6]. The approach is based on the transaction concept known from databases and allows programmers to specify transaction sequences which are executed atomically. In comparison to lock-based approaches, this concept provides:

- freedom from deadlock and priority inversion
- automatic roll-back on exceptions or timeouts
- freedom from the tension between lock granularity and concurrency

The approach is efficiently implemented as external C primitives in the newest release (6.4) of the Glasgow Haskell Compiler (*ghc*) [5]. The implementation relies on the fair implementation of Concurrent Haskell within ghc and is not portable to other implementation of Concurrent Haskell, as in Hugs [8].

We present an implementation of STMs within Concurrent Haskell, which is executable on every platform providing Concurrent Haskell, including Hugs (which also implements Concurrent Haskell within Haskell [2]). Although our (final) implementation is up to 6 times slower than the external implementation provided in ghc, it will be sufficiently fast in many applications (which usually do not perform transactions all the time as our benchmark programs do) and appears to be a good platform for experiments with possible extensions of STMs. A

high-level implementation has also the opportunity of being more maintainable. Finally, our completely different implementation could also inspire the developers of the external implementation for more elegant or even more efficient code.

The paper is organized as follows: Section 2 introduces `STM`s and Section 3 defines our basic `STM` monad definition. Our first implementation is defined in Section 4, which is then redefined to our second approach in Section 5. More implementation details are presented in Section 6, before we discuss benchmarks in Section 7 and conclude in Section 8.

2 Software Transactional Memory

Transactions provided by ghc 6.4 and described in [6] provide a monad `STM` as an abstract data type for transactions. The execution of a transaction is guaranteed to be "atomically" with respect to other concurrently executed threads. `STM`s provide *optimistic synchronization*, which means transactions are interleaved with other transactions. A transaction is committed only if no other transaction has modified the memory its execution depended on. Otherwise, the transaction is restarted.

For communication inside the `STM` monad it provides transactional variables, in terms of the abstract data type `TVar`. The interface is defined as follows:

```
data STM a  -- abstract
instance Monad STM

-- Running STM computations
atomically :: STM a -> IO a
retry :: STM a
orElse :: STM a -> STM a -> STM a

-- Transactional variables
data TVar a  -- abstract
newTVar   :: a -> STM (TVar a)
readTVar  :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
```

Transactions are started within the `IO` monad by means of `atomically`[1]. When a transaction is finished, it is validated that the transaction was executed on a consistent system state, i.e. no other finished transition may have modified relevant parts of the system state in the meantime. In this case, the modifications of the transaction are committed. Otherwise, they are discarded and the transaction is re-executed. Accordingly, inconsistent transactions can also be detected by programmers and aborted manually by calling `retry`. The provided implementation of `retry` avoids busy waiting and suspends the thread performing `retry` until a re-execution again makes sense. On top of `retry`, as a kind of

[1] In [6] this function was called `atomic`.

alternative composition, it is possible to combine transaction as: stm_1 `` `orElse` `` stm_2. If stm_1 performs a `retry` action, then this is caught and stm_2 is executed. If stm_1 succeeds, then stm_2 is not executed at all.

Data modifiable by transactions is stored in `TVar`s, which can only be manipulated within the `STM` monad. Beside modifications of `TVar`s, no other side effects like input/output are allowed with the `STM` monad, which makes it possible to re-execute transitions.

Finally, the `STM` monad provides exception handling, similar to the exception handling ghc provides for the `IO` monad. For details see [6].

As a simple example, we present an implementation of the well-known dining philosophers using `STM`s. The sticks are represented by boolean `TVar`s. `True` means the stick is laying on the table, i.e. it is available.

```
import STM

type Stick = TVar ()

takeStick :: Stick -> STM ()
takeStick s = do b <- readTVar s
                 if b then writeTVar s False
                      else retry

putStick :: Stick -> STM ()
putStick s =  writeTVar s True

phil :: Int -> Stick -> Stick -> IO ()
phil n l r = do atomically $ do takeStick l
                                takeStick r
                putStrLn (show n++". Phil is eating.")
                atomically $ do putStick l
                                putStick r
                phil n l r

startPhils :: Int -> IO ()
startPhils n = do sync <- newEmptyMVar
                  ioSticks <- atomically $ do
                    sticks <- mapM (const (newTVar True)) [1..n]
                    return sticks
                  mapM_ (\(l,r,i)->forkIO (phil eatings sync i l r))
                    (zip3 ioSticks (tail ioSticks) [1..n-1])
                  phil n (last ioSticks) (head ioSticks)
```

When trying to take a non-available stick, `takeStick` performs `retry`. The philosopher suspends until its neighbor puts the stick back onto the table. In the definition of `putStick` we do not perform a similar check since the behavior of one philosopher thread guarantees that the stick is not laying on the table (the `TVar` contains `False`) when performing `putStick`. However, this would be possible, too, and we used such a version for benchmarking as well. By combining the two actions for taking the sticks as one atomic `STM` transaction, the program

is deadlock free. Putting the sticks back on the table in one atomic action is not necessary, but is shorter than writing `atomically` twice.

The code for starting n philosophers is just presented for completeness and not discussed any further.

3 Implementing STMs in Concurrent Haskell

We present two different implementations of `STM`s within Concurrent Haskell. Both can be used in any Concurrent Haskell implementation, like older ghc versions and Hugs. In comparison to the STM implementation available in ghc 6.4, our implementations have a (worst-case) slow down between 4 to 30.

In both approaches we define the `STM` monad as an extension of the `IO` monad with a state used for collecting information about the execution of a transaction. The `STM` monad is defined similarly to other `IO` monad extension, e.g. the `GUI` monad defined in TclHaskell [3] or other libraries for graphical user interfaces:

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

instance Monad STM where
  (STM tr1) >>= k = STM (\state -> do
                           stmRes <- tr1 state
                           case stmRes of
                             Success newState a ->
                               let (STM tr2) = k a in
                                 tr2 newState
                             Retry newState -> return (Retry newState)
                             Invalid -> return Invalid
                         )
  return x = STM (\state -> return (Success state x))

data STMResult a = Retry STMState
                 | Invalid
                 | Success STMState a
```

The data type `STMResult` covers the relevant results of an `STM` action. The type `STMState` is the state carried through the `STM` monad. Its concrete realization will be discussed further in each implementation.

4 Collecting modifications in `TVars`

Our first approach for implementing `STM`s is closely related to the implementation presented in [6] where a thread-local transaction log is used to collect and share all `TVar` accesses within a thread. The transaction log holds references to all used TVars and is responsible for the necessary verify and commit actions.

The original `STM` implementation [6] uses thread-local `TLog`s referencing the global `TVar`s. Therefore, each `TLog` is a list pointing to each `TVar` used in its thread. Unfortunately, in general, `TVar`s have different types. This prohibits using

the same data structure in our implementation because of Haskell's strict type system. We use the opposite referencing structure instead: each `TVar` contains a log which simply stores a list of its new local values of all threads/atomic blocks it was modified by. This list maps transaction identifiers (`ID`, discussed in more detail in Section 6.2) to locally modified values.

The `TVar` value itself is stored as an `IORef` so that it can later be compared with the `TVar` value itself (`IORef (IORef a)`) by pointer comparison. Whenever a value is written to a `TVar` in a thread, we create a new `IORef` so that the comparison with the old value `IORef` will indicate the modification. Finally, the `TVar` contains a wait queue for retries (discussed in Section 4.1):

```
data TVar a = TVar (IORef (IORef a)) -- the TVar content
              (MVar [(ID,IORef a)]   -- thread local TVar copies
              (MVar [MVar ()])       -- wait queue on retry
```

If a `readTVar` or `writeTVar` action is performed, the corresponding thread updates its local version of the `TVar` value. To guarantee atomic modifications of the `TVar` copies, these are embedded into an `MVar`. The local `TVar` value is accessed by a transaction identifier. Here we first used `ThreadId`s to emulate the thread-local `TLog` structure used in [6]. `ThreadID`s can be used to identify each transaction as each thread can execute only one atomic transaction at a time. Since `ThreadId`s are not available in Hugs each `STM` action obtains a fresh `stmId` when started instead. Details how we provided identifiers are discussed in Section 6.2. The `stmId` is part of the `STMState` passed through our `STM` monad. Writing a `TVar` can now easily be defined as follows:

```
writeTVar (TVar _ tLog _) v = STM $ \stmState -> do
  tLogs <- takeMVar tLog
  putMVar tLog ((stmId stmState,v):tLogs)

readTVar (TVar tVarRef tLog _) = STM $ \stmState -> do
  tLogs <- takeMVar tLog
  case lookup (stmId stmState) tLogs of
    Just v -> return v
    Nothing -> do tVarVal <- (readIORef tVarRef)
                  readIORef tVarVal
```

This implementation simply maskes old values within the log list. In the real implementation these values are replaced, to keep log lists short.

Now, we have to find an implementation for checking the validity of a transaction and committing all modifications performed within a transaction. Again, the type system does not allow holding a list of all read (for checking validity) and written (for committing) `TVar`s. As a solution, we collect respective `IO` functions and eventually execute them at the end of the atomic block. For validating, this is an action of type `IO Bool` and for committing of type `IO ()`. Furthermore, we need a function for discarding the logs within the `TVar`s.

So far, we have introduced most of the information to be kept in the `STMState`, which is defined as the following record:

```
data STMState = STMState {isValid   :: IO Bool,
                          commit    :: IO (),
                          discard   :: IO (),
                          wait      :: IO (),
                          retryMVar :: MVar (),
                          stmId     :: ID}
```

The components `wait` and `retryMVar` are needed for suspending in `retry` and are discussed in Section 4.1.

Validation, commit and discard actions within the state are extended in each `readTVar` or `writeTVar` action. The function `readTVar` extends the already stored validation by a comparison of its own value with the local new value stored in its log:

```
readTVar (TVar tVarRef tLog _) = STM $ \stmState -> do
  ... -- read the value of the TVar bound in variable val
  oldVal <- readIORef tVarRef
    let newState = stmState{isValid  = do
                                b <- isValid stmState
                                if b then do
                                       tVarVal <- readIORef tVarRef
                                       return (tVarVal == oldVal)
                                     else return False}
  return (Success newState val)
```

In a `writeTVar` action the commit function is build up to copy the local `TVar` value into the real `TVar` thus committing a successful atomic action.

```
writeTVar tVar@(TVar tVarRef tLog _) = STM $ \stmState -> do
  ...
  let newState = stmState{
        commit  = do commitAct (stmId stmState) tVar
                     notify waitQ
                     commit stmState,
        discard = do tLogs <- takeMVar tLog
                     let (pres, _:posts) = break ((stmId==) . fst) tLogs
                     putMVar tLog (pres ++ posts)
                     discard stmState}
  return (Success newState ())

commitAct :: ID -> TVar a -> IO ()
commitAct stmId (TVar tVarRef tLog _) = do
  tLogs <- readMVar tLog
  tVarVal <- readIORef tVarRef
  let Just newRef = lookup stmId tLogs
  readIORef newRef
  newTVarVal <- newIORef v
  writeIORef tVarRef newTVarVal
```

The action `commit` does not only copy new values within the `TVars`. It also notifies other transactions suspended within a `retry` action. This and the implementation of `notify` are discussed in more detail in Section 4.1.

The function `atomically` starts and validates transactions. Non-valid transactions are discarded (the corresponding `TVar` copies are deleted) and restarted. Valid transitions are committed, i.e. the original `TVar` value is overwritten by the value stored for the actual `stmId`. The `IO` functions for discarding and committing values are constructed in the `readTVar` and `writeTVar` actions and stored in the `STMState` as shown above. The whole process of validating, committing and discarding may not be interupted by any other concurrent thread which is guaranteed by calling the functions `takeGlobalLock` and `freeGlobalLock`. Their implementation is discussed in Section 6.1.

```
atomically :: STM a -> IO a
atomically stmAction = do
  actionId <- getGlobalId
  stmResult <- startSTM stmAction actionId
  case stmResult of
    Invalid -> atomically stmAction
    Success newSTMState res -> do takeGlobalLock
                                  valid <- isValid newSTMState
                                  if valid then do
                                    commit newSTMState
                                    discard newSTMState
                                    freeGlobalLock
                                    return res
                                  else do
                                    discard newSTMState
                                    freeGlobalLock
                                    atomically stmAction
```

So far, a distinction between `Invalid` and `Retry` actions is not necessary and we only consider the `STMResult Invalid`. The discussion of `orElse` in Section 6.3 will distinguish these two cases.

4.1 Retry

In `STM`s it is also possible that the programmer marks a branch of the execution as invalid, by executing the `retry` action. For instance, a dining philosopher calls `retry` when trying to take a non-available stick. Of course, if he already sucessfully took his first stick, then he should put back his first stick, again. Naturally, this is implemented within an atomic action as shown before.

When implementing `retry` it does not make sense to directly restart the transaction, because the computation would execute exactly the same transaction again and deterministically reach the same `retry` action again. Its deterministic behavior only depends on the values of the `TVar`s it read during its execution. Hence, `retry` should suspend until any of the `TVar`s read during its execution is modified by another thread.

In Concurrent Haskell a thread can only suspend on no more than one `MVar`. Hence, we introduce a `retryMVar :: MVar ()` in the `STMState`, on which it suspends in `retry`. Again, we guarantee the atomic execution of validation, commit and restore by a global lock:

```
retry :: STM a
retry = STM (\stmState -> do
          takeGlobalLock
          valid <- isValid stmState
          discard stmState
          freeGlobalLock
          if valid then do wait stmState
                           takeMVar (retryMVar stmState)
                           return Invalid
                   else return Invalid)
```

After validating a transaction stored modifications are discarded. Then, in case of a valid transaction the thread should suspend. However, beforehand, it has to register itself for being awoken again in each read `TVar`. This is implemented by means of an accumulated `wait` action extending wait queues in all read `TVars` in a similar way as presented for `isValid`, `commit`, and `discard`:

```
readTVar (TVar tVarRef tLog waitQ) = STM $ \stmState -> do
  ... -- val is bound here
  let newState = stmState{wait = do
                           wait stmState
                           queue <- takeMVar waitQ
                           putMVar waitQ (retryMVar stmState:queue)}
  return (Success newState val)
```

After executing all the `wait` actions `retry` suspends on its `retryMVar` and after being awoken returns `Invalid` which initiates restarting the transaction in the enclosing `atomically`.

For awaking suspended transactions, each committed `writeTVar` action sets all registered `retryMVars`. The call to this notification was already integrated in the definition of the `commit` action in `writeTVar`. For completeness, we present the missing definition of `notify`:

```
notify :: MVar [MVar ()] -> IO ()
notify waitQ = do
  queue <- takeMVar waitQ
  mapM_ tryPutMVar queue
  putMVar waitQ []
```

5 The Collecting Approach

Profiling programs using the presented `STM` implementation shows that

- much time is spent for modifying `TVars` and
- validation, commit and notification are very fast.

Since collecting actions performs very well, it would be nice to extend this idea for the modifications while reading and writing as well. Inspecting transactions from a more abstract point of view we observe that

– reading is in most cases performed on original TVars and
– writing is delayed to copying in commit.

Hence, why don't we collect the writeTVar actions themselves in the commit actions instead of an action which copies a value within the TVar? Then, the only problematic case would be reading a TVar written beforehand. A modification of a TVar is only available after performing the commit.

On one hand, in practice, programmers will try to avoid reading an already written TVar, since the value written to the TVar is already known within the transaction. On the other hand, composing transactions may create such cases, in which two composed transactions modify and read the same TVar more than once. We assume, that this case may occur in practice (and has to be handled correctly by our implementation), but that it is not the regular case for every TVar in every transaction. We assume, the loss of efficiency for this special case will not matter compared to the gained speedup of using collected actions instead of modifying data structures within the TVar representations.

But how can we correctly access the value of an already modified TVar. The problem is, that we may not modify TVars to obtain the actual value, but the only place where this value is stored is the accumulated commit action. A solution is motivated by the search for deadlocks in Concurrent Haskell programs in [1]. Modifications of communication abstraction can be reversed. Hence, in parallel to accumulating the commit action, we accumulate a restore action. With these two actions, we can solve the problem with reading already written TVars. After setting a global lock and checking validity of the actual transaction, we can commit all TVar modifications, read the actual value, and revert the modified TVars. Fortunately, the overhead for this expensive operation will in some cases be balanced by the fact that the earlier validation restarts the transaction earlier.

To identify the situation in which an already written TVar is read, we have to collect all modified TVars within the STMState. Again, Haskell's type system does not allow such a data structure. As before, this problem can be solved by using unique identifiers which in this case identify the different TVars:

```
data TVar a = TVar (IORef (IORef a)) -- the TVar content
                   ID                -- TVar identifier
                   (MVar [MVar ()])  -- wait queue on retry
```

The identifiers of all modified TVars within a transaction are stored in the STMState. The modified STMState is defined as follows:

```
data STMState = STMState {stmId         :: IORef (),
                          modifiedTVars :: [IORef ()],
                          isValid       :: IO Bool,
                          commit        :: IO (),
                          notify        :: IO (),
                          restore       :: IO (),
                          wait          :: IO (),
                          retryMVar     :: MVar ()}
```

If such an already modified `TVar` is read, then we have to consider the "actual" value of this `TVar` within our `STM` Monad. Otherwise, `readTVar` behaves as before.

```
readTVar (TVar tVarRef tId waitQ) = STM $ \stmState ->
  if elem tId (modifiedTVars stmState) then do
     takeGlobalLock
     valid <- fIsValid
     if valid then do commit stmState
                      tVarVal <- readMVar tVarRef
                      val <- readIORef tVarVal
                      restore stmState
                      freeGlobalLock
                      return (Success stmState val)
              else do freeGlobalLock
                      return Invalid
  else ...
```

For completeness, we also present the new `writeTVar` code. The modifications of the `commit` and `notify` actions stay unchanged.

```
writeTVar (TVar tVarRef tId waitQ) v = STM $ \stmState -> do
  tVarVal <- readMVar tVarRef
  let newState =
       stmState{modifiedTVars=n:modifiedTVars threadState,
                commit = ...
                notify = ...
                restore = do takeMVar tVarRef
                             putMVar tVarRef tVarVal
                             restore stmState}
 return (Success newState ())
```

Note that we have to guard the commit and restore actions within `readTVar` described above by a validation check to ensure consistency.

Within the implementation details of [6] a potential problem arising from inconsistency has been highlighted with the following example:

```
f :: Integer -> Bool
f x = if x==0 then True else f (x-1)
foo v = atomically $ do
          x <- readTVar v
          y <- readTVar v
          if f (x-y) then ... else ...
```

An inconsistent view of `v` can lead to nontermination. The solution proposed in [6] is to check for consistency whenever the scheduler is about to switch a thread engaged in a transaction. Of course, with our high-level approach access to the scheduler is not easily feasable. Fortunately, this problem is similar to the problem of reading an already written `Tvar`. The solution is easy. We extend the `modifiedTVars` of a transaction to a list of `touchedTVars` which is also built up when reading a `TVar`. Then, an additional validity check can be started when

reading a `TVar` for the second time. For long transactions taking many schedule switches this may perform better than the approach taken in [6].

The nice idea behind the collecting approach is accumulating the whole transaction within IO actions, which may then be performed when reaching the end of the transaction. Benchmarks show that this implementation performs very well, as we will discuss in Section 7.

6 More Implementation Details

So far, we presented the whole implementation of `STM`s in Concurrent Haskell. However, some aspects of the implementation are not discussed yet.

6.1 Global Locks

In the presented implementations, we had to ensure that in some cases threads do not interfere, e.g., for validating and committing transactions. In the presented code, we called functions `takeGlobalLock` and `freeGlobalLock` in the corresponding cases. The simplest implementation of a global lock can be implemented as a global `MVar ()` constant by means of `unsafePerformIO`:

```
globalLock :: MVar ()
globalLock = unsafePerformIO (newMVar ())

takeGlobalLock :: IO ()
takeGlobalLock = takeMVar globalLock

freeGlobalLock :: IO ()
freeGlobalLock = putMVar globalLock ()
```

However, it is also possible to avoid this `unsafePerformIO` call. We add both a `lock` and an `unlock` `IO ()` action to the `STMState` and extend each `TVar` with an additional `MVar ()`.

```
data STMState = STMState{...
                         touchedTVars :: [ID],
                         lock         :: IO (),
                         unLock       :: IO ()}

data TVar a = TVar (MVar (IORef a))  -- global TVar itself
                   ID                -- TVar identifier
                   (MVar ())         -- TVar lock
                   (MVar [MVar ()])  -- wait queue on retry
```

Then the lock action is simply a collection of `takeMVar` calls on the `TVar`'s `MVar`. Likewise, the unlock action is performed by `putMVar` calls. In order to avoid deadlocks we must take care of collecting exactly one lock and unlock action for each `TVar` accessed. Therefore, we use only the first `readTVar` or `writeTVar`

call to add both lock and unlock calls for the accessed `TVar`. To identify this case, we can use the list of `touchedTVars` discussed in the end of Section 5. The `writeTVar` function works similarly.

```
readTVar (TVar tVarRef tId tVarLock waitQ) = STM $ \stmState -> do
  if elem tId (touchedTVars stmState)
    then ...
    else do
      ...
      let newState = stmState{wait = ...
                              touchedTVars = tId:touchedTVars stmState,
                              lock = do takeMVar tVarLock
                                        lock stmState,
                              unLock = do putMVar tVarLock ()
                                          unLock stmState}
      return (Success newState val)
```

However, measurements show that this implementation is slower than using a global lock. Usually, the time for which a global lock is set is very short and hence, we use a global lock in our implementation.

6.2 Unique Identifiers

In both approaches we needed unique identifiers: for identifying different `STM`s in the first approach and different `TVars` in the second approach.

Again, a simple implementation uses a global state of `Integer` values (defined by `unsafePerformIO`) which can only be increased, when getting a new id.

```
type ID = Integer

globalCount :: MVar ID
globalCount  = unsafePerformIO (newMVar [0..])

getGlobalId :: IO ID
getGlobalId = do
  num <- takeMVar globalCount
  putMVar globalCount (num+1)
  return num
```

Again, it would be nice to have an implementation without `unsafePerformIO`. The idea is to use `IORef`s instead of numbers, since it is possible to compare them.

```
type ID = IORef ()

getGlobalId :: IO ID
getGlobalId = newIORef ()
```

The garbage collector takes care of unused `IORef`s. There is no need for explicit releasing of identifiers at the end of an atomic block. However, the convenience of these runtime system provided identifiers has to be paid by a slight slow down.

An alternative `unsafePerformIO` free implementation of unique identifiers can be obtained by using stable pointers of Haskell's Foreign Function Interface [4, 11]. However, being able to avoid using `unsafePerformIO` is nice and shows the elegance of the presented implementation.

6.3 OrElse

So far, we have not considered the implementation of `orElse`. The semantics of combining two transaction as stm_1 `` `orElse` `` stm_2 is, that if stm_1 performs a retry action, then all modifications within stm_1 are discarded and stm_2 is performed. Hence, validating the whole transaction means validating that stm_1 is still valid (reaching `retry`) and stm_2 is valid. However, accumulated commit/restore actions within stm_1 have to be discarded. We implement this behavior by extending `commit`, `restore`, and `notify` to lists (stacks) of `IO` actions, the other parts of the `STMState` stay unchanged:

```
data STMState = STMState{...
                        commits  :: [IO ()],
                        notifys  :: [IO ()],
                        restores :: [IO ()],
                        ...}
```

At this point it is important to distinguish the `STMResults`:`Invalid` and `Retry`, as only in the latter `orElse` may execute the second transaction. In our second implementation `Invalid` as an intermediate `STMResult` is possible, since reading an already modified `TVar` performs an intermediate validation.

```
orElse :: STM a -> STM a -> STM a
orElse (STM stm1) (STM stm2) =
  STM (\stmState -> do
       stm1Res <- stm1 stmState{commits  = return ():commits stmState,
                                notifys  = return ():notifys stmState,
                                restores = return ():restores stmState}
       case stm1Res of
         Retry newState ->
              stm2 newState{commits = tail (commits newState),
                            notifys = tail (notifys newState),
                            restores = tail (restores newState)}
         _ -> return stm1Res)
```

Note, that `orElse` extends the list of commit/notify/restore actions when executing `stm1` and pops them again when (in case of `retry`) `stm2` is executed. As a matter of course, executing these actions in the definition of `atomically` (and `readTVar` as well) must consider the list structure. For instance, instead of `commit stmState` we have to perform:

```
sequence_ (reverse (commits stmState))
```

Reverting the list is necessary, because earlier modifications are located deeper in the list. And the chronological order is important if the same `TVar` is written twice within a transaction.

6.4 Exceptions

Exception handling as proposed for `STM`s can be integrated into the presented implementations easily and is not discussed any further in this paper.

7 Benchmarks

Benchmarking concurrent programs using `STM`s is very difficult as execution times strongly depend on the actual scheduling and sometimes differ significantly from each other. This is also true for ghc 6.4's `STM` implementation itself. In addition the deviation depends on the test programs used as benchmark. Finally, our benchmarks perform mostly transactions. Therefore, small modifications (e.g., using a global state or `IORef`s for implementing `ID`s) can show unproportional impact. In real applications this will usually not be the case (real programs should also compute something). Hence, the execution time of transactions will be less relevant and most application will work fine with any correct implementation.

However, we want to present some conclusions of our benchmarks. The test suite we used was executed automatically over and over again. At the end the median execution time was calculated and compared. Thereby we found that the statistical deviation between different runs discussed before was much higher than any performance gain of the median execution time by an improved implementation. However, the deviation of the original implementation is significantly smaller. Further research has to be made to analyse these effects.

Although, the effort for reverting modifications in the second approach and possibly overwriting own modifications within the same transaction seems less efficient than the first approach, our measurements refute this expectation. Even in examples, which modify the same `TVar` within one transaction several times (e.g. a version of the dining philosophers in which a philosopher picks up and lays down his sticks several times within an atomic transaction before eating), the first approach is slower.

We made benchmark tests of the different evolution steps of our implementation including the two versions aforementioned and some additional deviations with several concurrent programs accessing heavily the shared resources `TVars`. Both regular and optimized versions were benchmarked using the ghc system. Optimization speeds up the results of our first approach five times, our second approach and its deviations by a factor of three and even the built-in solution shown in [6] benefits by doubling the performance.

More of an interest is the comparison between the ghc based built-in solution and our different approaches in optimized form. The C-library solution in [6] is, of course, the fastest followed by our second (collecting) group of approaches showing a slowdown of four to eight times compared to ghcs implementation. By far the least performing implementation is our first one using many `TVar` accesses. It is from ten to 30 times slower than the built-in one.

8 Conclusion

We have presented two re-implementations of software transactional memory within Concurrent Haskell. The first implementation is closer related to the implementation with ghc 6.4. Analyzing the run-time behavior of this implementation yields to a more high-level implementation accumulating commit and validation actions within the state inside the `STM` monad. Benchmarks show, that in worst case this implementation is between four to eight times slower than the implementation in ghc 6.4. However, in real applications this will usually not be a problem, since these programs will also perform other computations.

On the other hand, our implementation also has some advantages: Our library can be executed in any Concurrent Haskell implementation, including Hugs and older ghc versions. It works independently of the underlying scheduling model. It is an implementation in a high-level language, which can be maintained and extended more easily. Hence, it should be a good platform for further research on transaction based communication in Concurrent Haskell. The library is available from the first author's web page.

For future work, we want to investigate how software transactions could be extended and how they could be used for distributed programming. A good basis should be the implementation of Distributed Haskell [7], which extends Concurrent Haskell to a distributed setting.

References

1. Jan Christiansen and Frank Huch. Searching for deadlocks while debugging concurrent haskell programs. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, pages 28–39, New York, NY, USA, 2004. ACM Press.
2. Koen Claessen. A poor man's concurrency monad. *Journal of Functional Programming*, 9(3):313–323, 1999.
3. Chris Dornan. Tcl + Haskell = TclHaskell. In *Glasgow FP Group Workshop, Pitlochry, Scotland*, September 1998. see also http://www.dcs.gla.ac.uk/ nww/TkHaskell/TkHaskell.html.
4. Manuel Chakravarty (ed.). The haskell 98 foreign function interface 1.0: An addendum to the haskell 98 report. http://www.cse.unsw.edu.au/chak/haskell/ffi/.
5. The Glasgow Haskell compiler. http://www.haskell.org/ghc/.
6. Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM Press.
7. Frank Huch and Ulrich Norbisrath. Distributed programming in Haskell with ports. In *Proceedings of the 12th International Workshop on the Implementation of Functional Languages*, volume 2011 of *Lecture Notes in Computer Science*, pages 107–121, 2000.
8. The Haskell interpreter Hugs. http://www.haskell.org/hugs/.
9. Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. In *Engineering theories*

of software construction, Marktoberdorf Summer School 2000, NATO ASI Series. IOS Press, 2001.
10. Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, Florida, 21–24 1996.
11. Simon Marlow, Simon Peyton Jones, and Wolfgang Thaller. Extending the haskell foreign function interface with concurrency. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 57–68, Snowbird, Utah, USA, September 2004.