A Haskell-Implementation of STM Haskell with Early Conflict Detection

David Sabel

Computer Science Institute
Goethe-University Frankfurt am Main
sabel@ki.informatik.uni-frankfurt.de

Abstract: Software transactional memory treats operations on the shared memory in concurrent programs like data base transactions. STM Haskell integrates this approach in Haskell and provides combinators to build software transactions in a composable way. In previous work we introduced the process calculus CSHF as a formal model for a concurrent implementation of software transactional memory in Concurrent Haskell. The correctness of CSHF with respect to a specification semantics was shown by using techniques from programming languages semantics. In this paper we first describe the CSHF semantics informally to illustrate the ideas of our approach. Secondly, we report on the implementation of the CSHF semantics as a library for software transactional memory in Haskell. In difference to the implementation available in the Glasgow Haskell Compiler our approach performs conflict detection as early as a possible, does not use pointer equality to test for conflicts and it is written in Haskell and thus not part of the runtime system. Besides explaining the features of CSHF and comparing them to the approach taken in the Glasgow Haskell Compiler, we explain in detail our design choices of the implementation.

1 Introduction

Writing efficient and correct concurrent and/or parallel programs is a challenge of computer science since several decades, and due to modern hardware architectures and modern distributed software this task is getting more and more important. Concurrent programming is much more difficult than sequential programming, since shared resources must be protected against parallel access which may corrupt the state of the resources or lead to race conditions.

Traditional approaches to implement concurrent programs are locks, semaphores, monitors, etc. All these approaches have in common that they are lock-based, i.e. critical sections are protected by locks to prevent that more than one thread enters the critical section at the same time. However, lock-based programming is error-prone: setting or releasing the wrong locks may lead to deadlocks and race conditions, setting too many locks makes program execution inefficient (since sequential execution is unnecessarily enforced), setting too few locks may lead to inconsistent states. Debugging concurrent programs is also much more difficult than debugging sequential programs, since thread execution is non-deterministic (depending on the scheduler) and even printing a debug message may change the scheduling of threads.

A recent approach to lock-free concurrent programming is transactional memory [ST95, ST97, HMPH05, HMPH08] which occurs as software transactional memory (STM) if implemented as a library of the programming language and as hardware transactional memory if implemented in hardware. We concentrate on STM. In this approach the shared memory is treated like a data base, and memory accesses are programmed as transactions. The implementation (i.e. the transaction manager) guarantees that transaction execution is performed atomically and isolated, i.e. transactions are either executed as a whole or nothing of the transaction is executed (atomicity), and concurrent execution of transactions is invisible, i.e. the effects on the memory of all transactions are identical to the effects of a sequential evaluation of all transactions (isolation).

There are some differences between data base transactions and software transactions: data base transactions must ensure durability on permanent storage while this is not the case for software transactions. Software transactions are not a closed system like data bases, i.e. surrounding program code can interfere with transactions and try to observe or break internals of transaction execution. A further difference is that for software transactions there is no guarantee that their execution terminates. However, the transaction manager should ensure at least global progress also if nonterminating and terminating transactions are running.

Using STM for programming is convenient and easy, since no locks need to be set and the transaction manager is responsible for correct execution. However, implementing the STM library and the transaction manager is difficult and usually requires lock-based programming.

In the functional programming language Haskell an STM-library (so-called STM Haskell) was proposed in [HMPH05]. It is implemented as part of the Glasgow Haskell Compiler (GHC) and used by a lot of programmers. STM Haskell combines the features of Haskell's strong type system and the ideas of STM to ensure that transactional code can only interfere with other code on clearly defined interfaces: the type system forbids to execute arbitrary side-effecting code inside software transactions, i.e. only reading and writing the transactional memory is allowed. This ensures that transactions are reversible and thus can be rolled back and restarted. The implementation of STM Haskell in the GHC is deeply embedded in the runtime system and it is written in C, which makes it highly efficient. However, as a disadvantage the code is not portable to other implementations of Haskell.

An important question is whether an STM-library is implemented correctly or not. Moreover, there is also no common notion of what correctness of such an implementation means. Several correctness properties, mostly based on the trace of the memory accesses, were proposed (see e.g. [GK08]). However, these notions do not capture the whole program behavior.

In [SSS13] we showed correctness of an STM Haskell-implementation, using techniques and notions of programming languages semantics: we modeled the specification and the implementation of STM Haskell as process calculi SHF and CSHF and showed correctness of the translation ψ : SHF $\rightarrow$ CSHF w.r.t. a contextual program equivalence.

However, our implementation calculus is different from GHC's implementation in C, since GHC's implementation is not suitable for being modeled by a process calculus: an obstacle is that the implementation compares the content of the transactional memory w.r.t. pointer equality and another one are temporary checks which are performed by the scheduler. For these reasons we proposed another implementation of STM Haskell which uses registrations on transactional memory and sending notifications to conflicting transactions. We also implemented a prototype of our proposal (called SHFSTM) in Concurrent Haskell [PGF96] (using synchronizing variables (so-called `MVars`) for locking). In the meantime we improved the prototypical implementation and made it more efficient.

The goal of this paper is twofold: on the one hand we present our results from [SSS13] in an informal manner by describing the ideas behind the process calculi and the correctness result, but omitting the formal definitions and details. On the other hand, we report on our Haskell implementation and show several problems which had to be solved for the implementation.

Related Work. Besides the mentioned implementation of STM in the GHC, there are some other approaches of implementing an STM library for Haskell. An implementation of the approach in [HMPH05], but in Haskell not in C was done by Huch and Kupke in [HK06] which is not as fast as the implementation in the GHC, but portable to other Haskell compilers and interpreters. However, the STM library of [HK06] does not perform temporary validity-checks of the transaction log against the global content of the transactional memory which may lead to semantically incorrect results, if non-terminating transactions are involved (see Sect. 3.4 for more details).

Another implementation of STM Haskell was done by Du Bois in [Boi11] where the TL2 locking algorithm [DSS06] was implemented in Haskell. This algorithm uses a global clock for creating time stamps on the transactional memory. Comparing the time stamps decides validity of read- and write-accesses. Also this implementation does not handle nonterminating transactions correctly.

A further approach of implementing the TL2 locking algorithm as a library for STM Haskell is Twilight STM [BMT10], which extends STM by executing arbitrary side-effecting code after the commit phase of a transaction. However, this library changes the behavior of STM Haskell and uses different type signatures, which makes it hard to compare to the other approaches.

Outline. In Sect. 2 we give an overview of STM Haskell and we explain some details of its implementation in the GHC. In Sect. 3 we summarize the results of our investigation in [SSS13] for showing correctness of an alternative STM Haskell implementation, and discuss advantages and disadvantages of our approach compared to the approach taken in the GHC. In Sect. 4 we describe the implementation of our STM Haskell library SHFSTM which is based on the CSHF-semantics. We show and discuss some parts of the Haskell code, and report on problems which had to be solved to implement the CSHF-semantics in the functional programming language Haskell. We also present experimental results of testing several Haskell STM libraries. We conclude in Sect. 5.

2 STM Haskell: The API and GHC's Implementation

In this section we recall the programmer's interface to Haskell's approach for software transactional memory as proposed in [HMPH05] and available as the `stm`-package[1]. We then briefly explain the implementation of STM Haskell in the Glasgow Haskell Compiler.

2.1 STM Haskell

The data type `STM a` represents an STM-transaction with return type `a`. The programmer builds an STM transaction (of type `STM a`) by using primitive operations and operators for composition of transactions. Execution of the transaction is performed by

```
atomically :: STM a → IO a
```

which takes an STM-action and runs it as a side effecting computation, where the implementation ensures atomicity and isolation of the execution.

Transactional variables `TVar a` are the main data structure of STM Haskell. They behave like shared-memory mutual variables, but they can only be read or modified inside STM-transactions, which ensures a consistent access. Primitive operations on `TVar`s are:

- `newTVar :: a → STM (TVar a)` to create a new `TVar`,
- `readTVar :: TVar a → STM a` takes a `TVar` and returns its content, and
- `writeTVar :: TVar a → a → STM ()` takes a `TVar` and a new content and writes this content into the `TVar`.

For composing `STM`-transactions the `STM`-type is an instance of the `Monad`-class, i.e. the `return`-operator for lifting any expression into the `STM`-monad, the `>>=`-operator for sequential composition of `STM`-transactions, and also the `do`-notation are available.

STM Haskell additionally provides the primitive `retry :: STM a` which gives the programmer explicit control about restarting transactions: if `retry` occurs during transaction execution, then the whole transaction is stopped, its effects are made invisible, and the transaction is restarted. A further combinator for the composition of transactions is `orElse : STM a → STM a → STM a`. The transaction `orElse` t_1 t_2 first tries to execute transaction t_1. If this execution is successful, then `orElse` t_1 t_2 is successful with the same result. If the execution of t_1 ends with `retry`, then the transaction t_2 is executed: if t_2 ends successfully, then this result is the result of `orElse` t_1 t_2, and otherwise `orElse` t_1 t_2 retries. Note that `orElse`-calls can also be nested.

Finally, STM Haskell also includes exceptions and exception-handling for the `STM`-monad. Since our semantic investigation in [SSS13] does not include exceptions, we will ignore exceptions for the remainder of this paper and leave the inclusion of exceptions for future work.

[1] http://hackage.haskell.org/package/stm

2.2 The Implementation of STM Haskell in the GHC

The implementation of STM Haskell in the GHC is embedded in the runtime system and implemented in C. We explain how transaction execution is performed by GHC's transaction manager. We first omit the `orElse`-command and assume that there is no `orElse`, later we will also explain the implementation of `orElse`-execution.

Every transaction uses a local transaction log for book-keeping of the read and write-operations on `TVars`. For every accessed `TVar` the value of the old content and the value of the new content are stored in the log. For `TVars` which were only read these values are identical. Write operations update the new value in the transaction log, but not the content of the TVar. The old values are used for deciding whether the transaction log is conflicting: the transaction log is valid iff all old values are *pointer equal* to the current content of the TVar, otherwise the transaction log is conflicting. When all operations of the transactions are performed, the accessed `TVars` are locked and the transaction log is tested for validity. In case of a conflicting transaction, the locks are removed and the whole transaction is restarted (the rollback consists only of throwing away the current transaction log). Otherwise, if the transaction log is valid, the new content in the transaction log is copied into the `TVars` and the locks are removed.

If a `retry` occurs during transaction execution, a restart with an empty transaction log is performed. However, as an optimization the restart due to a `retry` command is delayed by waiting until any content of an accessed `TVar` changes, since otherwise the restarted transaction would end again with `retry`.

For implementing `orElse`, instead of storing a single new value in the transaction log, a stack of those values is stored. Execution of `orElse` adds a new level on all these stacks and aborting from the left argument of an `orElse`-evaluation by a `retry`-command removes the top-element on all stacks.

To detect conflicting transactions earlier, validity of the transaction log is not only checked at commit time, but also temporarily during thread execution. Later we will argue that this is not only an optimization, but also a requirement for the correct implementation of STM Haskell (see Sect. 3.4).

3 A Correct Implementation of STM Haskell

In [SSS13] we introduced two process calculi to model the semantics of Haskell STM: the calculus SHF is the *specification semantics*, i.e. it specifies the correct behavior of STM Haskell. The calculus CSHF is the *implementation semantics*, i.e. it defines a detailed implementation of STM Haskell by using transaction logs, concurrent execution of transactions, commit phases, and rollback and restart of transactions.

3.1 The Specification Semantics

The specification semantics SHF is similar to the semantics of STM Haskell given already in [HMPH05]. Differences are mainly that SHF is based on a model of Concurrent Haskell extended by futures [SSS11, Sab12] (which slightly extend threads of Concurrent Haskell in a safe manner [SSS12]), and that evaluation of purely functional expressions is defined for SHF using a call-by-need operational semantics which respects sharing and avoids code duplication, while the approach of [HMPH05] uses a (not specified) denotational semantics.

These semantics are specifications only, since they are not directly implementable: they include reduction rules whose applicability is not decidable (but semi-decidable). One of such rules is the following rule for transaction evaluation (which has further restrictions on the contexts C, C', C_1, C_1' described below), i.e. it defines the semantics of the `atomically`-operator:

$$\frac{C_1[\texttt{atomically}\ e] \xrightarrow{\mathsf{SHFA},*} C_1'[\texttt{atomically}\ (\texttt{return}_{\texttt{STM}}\ e')]}{C[\texttt{atomically}\ e] \xrightarrow{\mathsf{SHF}} C'[\texttt{return}_{\texttt{IO}}\ e]}$$

Here $\xrightarrow{\mathsf{SHF}}$ denotes the reduction relation of SHF and $\xrightarrow{\mathsf{SHFA},*}$ means a finite sequence of a special reduction relation, which only allows to evaluate transactions. The above rule can be read as follows: if `atomically` e can be evaluated with $\xrightarrow{\mathsf{SHFA}}$-steps successfully in the `STM`-monad (ending in a `return`), then the reduction $\xrightarrow{\mathsf{SHF}}$ can do this in one step, and returns the result in the `IO`-monad. The different contexts C, C', C_1, C_1' mean that C has its hole in a concurrent thread at reduction position and C_1 is the part of C which contains this single thread together with the transactional variables and heap bindings (but no other threads). The local evaluation of this thread and environment then results in the modified environment C_1' and these modifications describe the difference between C and C' in the original whole program.

The application of this big-step rule is not decidable, since one cannot decide whether the transaction evaluation halts. Moreover, another disadvantage of the specification semantics is that transactions are not evaluated concurrently, but in a sequential order, one transaction after the other. However, this perfectly matches the semantics of software transactions, since it guarantees isolation and atomicity of transaction evaluation.

3.2 The Implementation Semantics

The implementation semantics CSHF uses only small-step reduction rules. Application of all rules is decidable. In this section we describe the main ideas of the implementation semantics.

3.2.1 Syntactic Extensions

We first explain the syntactic extensions of CSHF w.r.t. STM Haskell:

Transactional Variables. CSHF uses two different kinds of transactional variables:

- So-called *global* `TVars` are globally accessible, mutable shared-memory cells which store the true content of the `TVar` and some further information:
  - a *locking label*, which allows to lock the `TVar` by a single thread: if the locking label of `TVar` a is set to thread identifier i, then only thread i can access a, all other threads which want to access the `TVar` are blocked. In the operational semantics of CSHF there is simply no matching reduction rule for the case that the `TVar` is locked by some other thread.
  - a set of thread identifiers of those threads which perform evaluation of an STM-transaction and have read the global content of the TVar. This set is called the *notify list* of the `TVar` and it is used for conflict detection.
- Every thread which performs evaluation of an STM-transaction has a local set of so-called *local* `TVars`. These are copies of global `TVars`, but the content is stored as a stack for `orElse`-evaluation. These local `TVars` are similar to the entries in the transaction log of GHC's implementation, but no old values are stored (see Sect. 2.2).

Transaction Log. Besides the associated set of local `TVars`, the transaction log consists of the following additional information which is stored for every thread i which performs evaluation of an STM-transaction.

- A set T of pointers to global `TVars` where those `TVars` are stored whose modification may lead to a conflict. This is a subset of all read `TVars` and exactly represents the global `TVars` which have the thread identifier i in their notify lists.
- A stack of triples (L_a, L_n, L_w) where
  - L_a is the set of accessed (read, written, and created) `TVars`,
  - L_n is the set of created `TVars`, and
  - L_w is the set of written `TVars`.

  These triples are stored in a stack, since several copies of the triples are needed for backtracking during evaluation of `orElse`. The set T is not stacked, since any conflicting read `TVar` results in a conflict for the transaction, independently from the `orElse`-branch where the `TVar` was read.
- A list K of `TVars` which is used during commit to store the `TVars` which where locked by the committing thread.

A further syntactic extension are special variants of the `atomically`- and the `orElse`-operator, which indicate active evaluation of the corresponding transactions. There is also a variant of `atomically` which additionally stores the original expression defining the transaction, which is necessary for restarting the transaction.

3.2.2 Operational Behavior

We describe the main ideas of the operational semantics of CSHF, which defines transaction execution. In our explanations we treat all operations on stacks as operations on the top element, unless otherwise stated.

Start of transaction execution. The sets T, L_a, L_w and L_n are initialized as empty sets.

Thread i creates a new `TVar` t. A thread local `TVar` is created and added to the sets L_a and L_n.

Thread i reads `TVar` t. The read-operation is performed on the local copy of t for thread i. If no copy exists (the read-operation is the first operation on t), then a copy for thread i is created *and* the thread identifier i is added to the notify list of t. The `TVar` t is also added to the sets L_a and T.

Thread i writes `TVar` t. The write-operation is performed on the local copy of t for thread i. If no copy exists (the write-operation is the first operation on t), then a copy for thread i is created. Note that i is *not added* to the notify list of t. The `TVar` t is also added to the sets L_a and L_w.

Thread i evaluates `retry` outside `orElse`. In this case the transaction must be restarted. Therefore thread i walks iteratively through its set T and removes its entry in the notify list of every `TVar` $t \in T$. Note that this removal is done in several steps and not in one global step (which would require to lock all `TVar`s in T). Thereafter the transaction restarts with sets T, L_a, L_n, L_w reset.

Thread i evaluates `orElse` e_1 e_2. Then for all local `TVar`s the top element of the content stack is duplicated, and for the triple stack also the top element (L_a, L_n, L_w) is duplicated. Then e_1 is evaluated.

Thread i evaluates `retry` inside `orElse`. In this case all stacks for local `TVar`s are popped, and also the triple stack is popped. Thereafter the second argument of `orElse` is evaluated. Note that the set T is not changed, since a conflict with a read `TVar` in another `orElse`-branch should also lead to a restart of the transaction.

Thread i commits. If all operations of a transaction are executed, then thread i starts its *commit phase*. As a first step all read and to-be-written `TVar`s are locked by thread i (by setting the locking label of the corresponding `TVar`s to i). This set of `TVar`s is computed as $T \cup (L_a \setminus L_n)$. In CSHF this locking is performed as an indivisible atomic step. To keep track of the locked `TVar`s, thread i stores the set $T \cup (L_a \setminus L_n)$ as the set K.

After setting the locks, other threads cannot access these `TVars`. Now the following steps are performed which may be interleaved by other transactions (accessing other `TVars`):

1. Thread i iteratively removes its entry in all notify lists of `TVars` $t \in T$ and removes t from T. This iteration stops when $T = \emptyset$.
2. Thread i iteratively walks over all notify lists corresponding to the `TVars` in L_w: if thread i' is member of such a notify list, then thread i sends a `retry`-command to thread i', i.e. it notifies i' to restart. In the operational semantics of CSHF this is specified by replacing the code of thread i' by `retry` (at the right position inside `atomically`). Thereafter thread i' performs in the same way as executing a programmed `retry` (first it removes its notify entries and then it restarts, as described above). The iteration stops when all notify lists of `TVars` in L_w are empty.
3. Thread i iteratively updates the global content of the `TVars` by copying the content of the local `TVars` into the global ones, for every `TVar` $t \in L_w$ (removing $t \in L_w$).
4. If $L_w = \emptyset$, then thread i iterates over the set K and removes its lock on every `TVar` t with $t \in K$.
5. Thread i iterates over the set L_n and creates a global `TVar` for every locally created `TVar` in L_n.
6. Thread i removes all local `TVars` and returns the result of transaction execution.

We argue why we claim that CSHF detects conflicts as early as possible: the CSHF semantics restarts conflicting transactions when the committing transaction determines a conflict (in step 2.). It is not possible to restart a transaction earlier: for instance one cannot restart a transaction i_1 at the moment when another transaction i_2 writes into the `TVar` t with i_1 in the notify list of t: although i_1 is conflicting when i_2 commits, there is no guarantee that i_2 executes its commit phase, and this may happen because i_2 runs into an infinite loop or i_2 is rolled back and restarted due to a conflict caused by another `TVar`.

3.3 Correctness of the Implementation

In [SSS13] we proved that CSHF is a correct implementation of SHF (and thus STM Haskell), by first defining a contextual semantics for both calculi which equates programs if their observable behavior is indistinguishable in any enclosing program context. Since both calculi are non-deterministic, the contextual equivalence uses two observations of the behavior: may-convergence $\downarrow$ holds for a program P, if the main-thread of P can be reduced into a `return` , and should-convergence $\Downarrow$ holds for P, if every reduction successor of P remains may-convergent.

Correctness of the implementation was shown by inspecting the translation ψ : SHF $\rightarrow$ CSHF which is more or less the identity. We have shown that ψ preserves and reflects may- and should-convergence, i.e. for every SHF-program P: $P\downarrow \iff \psi(P)\downarrow$ and $P \Downarrow \iff \psi(P)\Downarrow$. This shows that CSHF can be used to evaluate SHF-programs without any different observation. Hence, CSHF is a correct *interpreter* of SHF. Additionally, we have shown that ψ is adequate, i.e. if $\psi(P_1)$ and $\psi(P_2)$ are contextually equal in CSHF, then P_1 and P_2 are contextually equal in SHF. This e.g. implies that program transformations performed on CSHF-programs that preserve the program equivalence, are also correct program transformations for the SHF-semantics. I.e., a correct compiler for CSHF does not change the semantics viewed from SHF.

3.4 Comparison of CSHF and GHC's STM Implementation

We explain the main differences between the semantics specified by CSHF and the implementation of STM Haskell in the GHC. In GHC transactions test their transaction log against the current state of the `TVars` and thus recognize a conflict by themself, in CSHF the conflicting transactions are notified and restarted by the committing transaction. We will discuss the advantages and disadvantages of the CSHF-approach.

Advantages. Advantages of the CSHF approach are that conflict detection may occur earlier than in the GHC, and that CSHF does not rely on comparing contents of `TVars`. Moreover, in CSHF there is no need to temporarily check the transaction log against the state of the global storage. Note that those checks are necessary in GHC's implementation to ensure correctness w.r.t. the STM Haskell semantics (as e.g. given by SHF): consider the following Haskell code which runs two transactions `i1` and `i2` which access the `TVar tv`:

```
i1 tv = atomically $ do                  i2 tv = atomically (writeTVar tv False)
  c ← readTVar tv
  if c then                              main = do tv ← atomically (newTVar True)
   let loop = do loop in loop                      s ← newEmptyMVar
  else return ()                                   forkIO (i1 tv ≫ putMVar s ())
                                                   threadDelay 10000
                                                   forkIO (i2 tv ≫ putMVar s ())
                                                   takeMVar s ≫ takeMVar s
```

Note that the `MVar` is used to wait for termination of both concurrent threads `i1` and `i2`. Execution of `main` should terminate successfully in any case: suppose that first i_1 is executed, and since it reads `True` for `tv`, it loops. If transaction i_2 now runs and commits, then in CSHF, transaction i_1 is notified, restarted, and in the next run it will terminate successfully. In GHC it is necessary that i_1 checks its transaction log against the global content of the `TVars` even it does not reach its commit phase. In the specification semantics SHF the transaction i_1 can only be performed (as a single big-step reduction) if the content of `tv` is already set to `False` which enforces to evaluate first `i2` and then `i1` and thus the whole program terminates in any case.

Testing the program in GHC shows that the program does not behave as expected, since GHC detects the loop, throws an exception and finally blocks forever on the MVar. If we change the code for the loop into `let loop i = do loop (i+1) in loop 1`, then the program behaves as expected, since the loop is not detected, and so after enough time the transaction `i1` is restarted. However, this shows that loop-detection does not work well together with the temporary check of the transaction log and perhaps should be turned off for transaction execution. The Haskell STM implementations in [HK06, Boi11, BMT10] all perform only a validity check during the commit phase and hence do not terminate in both cases, which can also be observed by testing the program with these libraries.

Considering nonterminating transactions, which always do not terminate, and which do not read any `TVar`, e.g. a transaction like

```
atomically $ let loop = loop in if loop then return 1 else return 2
```

the "operational" behavior of the specification semantics is different from all STM Haskell implementations: In SHF the this transaction is never executed, while the implementations (including CSHF) start the execution, but never detect a conflict nor reach the commit phase. However, *semantically* this is correct, since the nonterminating transaction is equivalent to a nonterminating program $\bot$, and SHF simply does not touch the $\bot$, while the implementations evaluate $\bot$, which makes no difference.

Disadvantages. A disadvantage of CSHF compared to the GHC approach is that rollback of transactions in CSHF requires to remove the entries of the thread in all notify lists while in GHC it is sufficient to throw away the transaction log. A further disadvantage of CSHF is that there a few more cases where a conflict occurs: if a transaction i_1 reads the content c of a `TVar` t, then transaction i_2 reads t and finally i_1 writes back the same content c into t and i_1 commits, then the transaction i_2 is restarted in CSHF. In contrast, since GHC uses a pointer equality test, the transaction log of i_2 would still be valid and no restart happens. However, these cases may be rare and e.g. the following transaction for i_1:

```
  atomically $ readTVar tv >>= λc → writeTVar tv (let x = c in x)
```

makes the transaction log of i_2 invalid, since pointer equality of `c` and `(let x = c in x)` does not hold.

4 The Implementation of CSHF in Haskell

In this section we explain our implementation of the CSHF-semantics as the SHFSTM-library in Haskell which is available from

http://www.ki.cs.uni-frankfurt.de/research/stm.

4.1 Implementing Local and Global `TVars` and the Transaction Log

While the CSHF-semantics associates the set of local `TVars` to its thread, this approach does not work well for an implementation in Haskell. The set would be of heterogeneous type, since `TVars` with different content type must be stored. It is also not possible to use existential types to hide the content types in the set, since e.g. the readTVar-operation would not be typeable, since it extracts the content type. To avoid this problem, our implementation uses an associated map for every global TVar which stores the local content stack for every running transaction.

The triple stack and the sets T and K are also heterogeneous sets of TVars. For implementing these sets there are at least the following three possibilities:

- The triple stack, T and K are implemented as heterogeneous sets using existential types. This is a typical use case for existential types, since all corresponding operations on the stack and the sets do not make visible the content type of TVars (they look into notify lists, they copy from local to global content, or they set locks).
- Instead of storing the TVars heterogeneously, we could store the monadic actions (inspecting the notify lists, copying etc.). This approach was used in [HK06].
- We could convert the TVars into pointers and cast them back for the needed operations (using unsafeCoerce). This approach was used in [Boi11].

The second approach is very elegant, since it does not require extensions of the Haskell standard, however, it makes the program harder to understand. The third approach is similar to the first one, where instead of using a type system extension unsafe operations are used. Thus we prefer the first choice, mainly since it is the closest one to our formal specification.

Our implementation thus uses *two* references to transactional variables: one with a polymorphic type and one with a monomorphic type (where the content type is hidden using existential types). Thus, a TVar a is a pair of a TVarA a (the polymorphically typed TVar) and a TVarAny (where the content type is hidden). Both types are MVars (for synchronizing the accesses) which contain the data of type ITVar a (explained below).

Additionally, since we use sets of TVarAny, an Ord-instance is required and thus we add unique identifiers TVarId to TVarAny, which are Integers:

```
newtype TVar a  = TVar (TVarA a,TVarAny)
newtype TVarA a = TVarA (MVar (ITVar a))
data TVarAny    = ∀ a. TVarAny (TVarId, MVar (ITVar a))
```

An invariant of our implementation of TVars is that both MVars always point to the same object. Now we explain the data type ITVar. As said before, we store the global content and a map which holds the local content for every thread identifier. All these components are protected by MVars to make them mutable and to protect them from parallel accesses. The third components of an ITVar is the notify-list which holds a set of thread identifiers, the fourth component represents the lock which is used during the commit phase, and there is one more component – called waiting queue – which implements an optimization: if a thread wants to access a locked TVar it waits on the waiting queue until the committing threads wakes it up again. Hence the type ITVar is defined as the following record type:

```
data ITVar a = TV { globalContent :: MVar a
                  , localContent  :: MVar (Map.Map ThreadId (IORef [a]))
                  , notifyList    :: MVar (Set.Set ThreadId)
                  , lock          :: MVar ThreadId
                  , waitingQueue  :: MVar [MVar ()] }
```

The transaction log of a transaction consists of the set T (called `readTVars`), the stack of triples (L_a, L_n, L_w) (called `tripelStack`) and the set K (called `lockingSet`). These sets are implemented as sets (a stack of triples of sets, resp.) of `TVarAny`, which allows to store the heterogeneous sets. To allow easy updates the log is stored in an `IORef`:

```
data Log  = Log { readTVars   :: Set TVarAny,
                , tripelStack :: [(Set TVarAny,Set TVarAny,Set TVarAny)]
                , lockingSet  :: Set TVarAny                              }
newtype TLOG = TLOG (IORef Log)
```

4.2 Implementation of the STM Monad and `atomically`

The data type `STM` implements data constructors for all STM-operations, where continuations corresponding to the ≫=-operator for sequencing are packed into the constructors.

```
data STM a = Return a
           | Retry
           | ∀ b. NewTVar b (TVar b → STM a)
           | ∀ b. ReadTVar  (TVar b) (b → STM a)
           | ∀ b. WriteTVar (TVar b) b (STM a)
           | ∀ b. OrElse (STM b) (STM b) (b → STM a)
```

The STM-operations generate the corresponding data, and the monad instance for the type `STM` is implemented straightforward, where the bind-operator ≫= combines the actions:

```
newTVar :: a  → STM (TVar a)
newTVar a = NewTVar a return

readTVar :: TVar a → STM a
readTVar a = ReadTVar a return

retry :: STM a
retry = Retry

writeTVar :: TVar a → a → STM ()
writeTVar v x = WriteTVar v x (return ())

orElse :: STM a → STM a → STM a
orElse a1 a2 = OrElse a1 a2 return

instance Monad STM where
 return  = Return
 m≫=f = case m of  Return x         → f x
                    Retry            → Retry
                    NewTVar x ct     → NewTVar x (λi → ct i≫=f)
                    ReadTVar x ct    → ReadTVar x (λi → ct i≫=f)
                    WriteTVar v x ct → WriteTVar v x (ct≫=f)
                    OrElse a1 a2 ct  → OrElse a1 a2 (λi → ct i≫=f)
```

A program of type STM thus is represented by a data structure. It is executed by applying atomically to it. As explained before, in CSHF, the committing thread sends notifications to conflicting threads and thus enforces their rollback and restart. We implement this behavior using exceptions: every transaction has an exception handler that handles the exception for retry, and the committing thread will throw a synchronous exception to conflicting threads. We introduce a new exception type RetryException for the thrown exception. The implementation of atomically is as follows: it creates a new transaction log, calls the function performSTM, which will execute the transaction, and catches the retry exception. If the exception is caught, first the function globalRetry is called, which removes all notify entries of the thread in accessed TVars, and then an atomically-call restarts the transaction execution. This sequence of actions is protected against receiving further exceptions using uninterruptibleMask_:

```
atomically :: STM a → IO a
atomically act = do
 tlog ← emptyTLOG
 catch (performSTM tlog act)
  (λe → case e of
         RetryException →
              (uninterruptibleMask_ (globalRetry tlog)) ≫ atomically act
         _ → throw e)
```

The function performSTM calls a function for every action of the STM transaction:

```
performSTM tlog act  =
 case act of
  Return a          → commit tlog ≫ return a
  Retry             → waitForExternalRetry
  NewTVar x ct      → newTVarWithLog tlog x  ≫= λtv → performSTM tlog (ct tv)
  ReadTVar x ct     → readTVarWithLog tlog x ≫= λc → performSTM tlog (ct c)
  WriteTVar v x ct  → writeTVarWithLog tlog v x ≫ performSTM tlog ct
  OrElse ac1 ac2 ct → do orElseWithLog tlog
                         resleft ← performOrElseLeft tlog ac1
                         case resleft of
                           Just a  → performSTM tlog (ct a)
                           Nothing → do orRetryWithLog tlog
                                        performSTM tlog (ac2 ≫= ct)
```

If Return a occurs, then it is the last action of the transaction and thus the commit phase has to be started, and thus the function commit (see Sect. 4.4) is called, before returning the resulting value in the IO-monad.

If a retry occurs, we could call the globalRetry-function in the same way as we do when a RetryException gets caught. However, if the content of the TVars is not changed, the execution of the transaction will result in retry again. That is why we call waitForExternalRetry, which simply blocks the thread on an MVar. If the content of accessed TVars changes, then the transaction becomes conflicting and will receive an exception, and otherwise if the content of the TVars never changes, then restarting the transaction will run in the same retry again. Thus, our implementation prefers blocking instead of busy-waiting. The same approach is used in GHC's STM Haskell implementation, but in our approach it is much simpler, since there is no need for additional waiting queues on `TVars` and additional signaling code.

For creating, reading, and writing `TVars` the special functions `newTVarWithLog`, `readTVarWithLog`, and `writeTVarWithLog` are called, which modify the transaction log and adjust the local copies of the `TVars`. We will explain only the `readTVarWithLog`-operation in the subsequent section.

For performing nested `orElse`-evaluation we use a special variant of `performSTM`, called `performOrElseLeft`, which either returns `Just`, if evaluation finishes with `Return`, or it returns `Nothing`, if evaluation ends with `Retry`.

4.3 Implementation of the Read-Operation

We explain the function `readTVarWithLog` that executes the read-operation during transaction evaluation. If the global `TVar` is locked, since another process is committing, the read-operation must not access the global `TVar` and should wait. Therefore the `readTVarWithLog`-function iteratively calls `tryReadTVarWithLog` until the read-operation was successful.

```
readTVarWithLog :: TLOG → TVar a → IO a
readTVarWithLog tlog tvar =
  do res ← tryReadTVarWithLog tlog tvar
  case res of Right r        → return r
              Left blockvar → takeMVar blockvar ≫ readTVarWithLog tlog tvar
```

If the `TVar` is locked, then the transaction adds a new `MVar` to the waiting queue of the `TVar` and waits to become unblocked (which is the responsibility of the committing thread). The code of `tryReadTVarWithLog` is shown in Fig. 1. We explain the code: if a local copy of the `TVar` exists (determined by inspecting the set L_a (lines 4–7)), then the local content is extracted and returned (lines 8–11). If no local copy exists, then `tryReadTVarWithLog` tests whether the `TVar` is locked (line 13). If the `TVar` is not locked, then the thread adds its identifier to the notify list, reads the global content of the `TVar`, creates the local copy of the content, and adjusts its transaction log (lines 15–26). If the `TVar` is locked, then the thread creates a new `MVar` and adds this `MVar` to the `waitingQueue`, and returns the `MVar` to block on (lines 28–32).

Every read-attempt is protected (using `uninterruptibleMask_`) against receiving the retry exception, since otherwise the concurrent data structures can become corrupted.

```
tryReadTVarWithLog (TLOG tlog) (TVar (TVarA tva,tvany)) =
 uninterruptibleMask_ $ do
  _tva ← takeMVar tva                          -- access the TVar
  lg ← readIORef tlog                          -- access transaction log
  let ((la,ln,lw):xs) = tripelStack lg
  mid ← myThreadId
  if tvany `Set.member` la then do             -- local TVar exists:
     localmap ← readMVar (localContent _tva)   -- local content
     contentStack ← readIORef (localmap mid)
     putMVar tva _tva
     return (Right (head contentStack))
   else do                                -- local TVar doesn't exist:
     b ← isEmptyMVar (lock _tva)
     if b then do                              -- TVar is unlocked:
        nl ← takeMVar (notifyList _tva)        -- add to notify list
        putMVar (notifyList _tva) (Set.insert mid nl)
        globalC ← readMVar (globalContent _tva) -- read global content
        content_local ← newIORef [globalC]
        mp ← takeMVar (localContent _tva)
        writeIORef tlog                        -- adjust transaction log
                   (lg{readTVars = Set.insert tvany (readTVars lg),
                       tripelStack = ((Set.insert tvany la,ln,lw):xs)})
        putMVar (localContent _tva)            -- copy to local
                (Map.insert mid content_local mp) -- TVar stack
        putMVar tva _tva
        return (Right globalC)                 -- return the content
      else do                                  -- TVar is locked:
        blockvar ← newEmptyMVar                --  add the thread to the
        wq ← takeMVar (waitingQueue _tva)      --  waiting queue
        putMVar (waitingQueue _tva) (blockvar:wq)
        putMVar tva _tva
        return (Left blockvar)                 -- return the blockvar
```

Figure 1: The implementation of `tryReadTVarWithLog`

4.4 Implementation of the Commit-Phase

The function `commit` performs committing of the transaction. It calls seven function, where the first function performs the locking of the read and to-be-written `TVars`.

The remaining six function calls correspond to the steps described in Sect. 3.2.2, i.e. removing the entries in the notify lists, sending the retry exception to conflicting threads, writing the local content into the global `TVars`, creating global `TVars` for the new `TVars`, removing the local `TVars`, and removing the locks. The implementation of `commit` is:

```
commit :: TLOG  → IO ()
commit tlog =
 do writeStartWithLog tlog  -- lock the TVars
    writeClearWithLog tlog  -- remove notify entries of the committing thread
    sendRetryWithLog tlog   -- notify conflicting threads
    writeTVWithLog tlog     -- copy local content into global TVars
    writeTVnWithLog tlog    -- create the new TVars
    writeEndWithLog tlog    -- clear the local TVar-stacks
    unlockTVWithLog tlog    -- unlock the TVars
```

Here the implementation is a little bit different from the CSHF-semantics: The unlock-phase is at the end, while in CSHF it is before creating the new `TVars`. The reason is that in CSHF newly created `TVars` cannot be accessed before the `TVars` are globally created (there is simply no binding, and threads cannot proceed without that binding), but in our implementation the `TVars` already exist (but should not be accessed). The earliest point of time when other threads may get references to the new `TVars` is after unlocking the modified `TVars`, since they may be updated by references to newly created `TVars`.

We do not explain the implementation of the subprocedures in detail, but discuss some interesting details about the first step, `writeStartWithLog`, and the third step, `sendRetryWithLog`.

4.4.1 Locking `TVars` During Commit

The CSHF-semantics locks the `TVars` in $T \cup (L_a \setminus L_n)$ by one reduction rule in one single indivisible step. In a previous prototype of our implementation we implemented this locking by using a global lock on all `TVars`, then locked the `TVars` in $T \cup (L_a \setminus L_n)$ and then removed the global lock. The performance of this approach was poor, since transactions became blocked for a while due to the global lock, even if they operated on disjoint sets of `TVars`. In our current implementation the phase of locking the `TVars` is no longer atomic. We lock the `TVars` one by one (without using a global lock) in ascending order of the to-be-locked `TVars`, which is possible since the type `TVar` has an `Ord`-instance. The total order theorem thus ensures that no deadlock can occur: if several transactions perform `writeStartWithLog` at least one transaction will lock all `TVars` for its corresponding set $T \cup (L_a \setminus L_n)$. However, we had to be careful with those transactions that do not get all locks. It is not correct to let them wait until the next lock becomes available: such a waiting transaction may be a conflicting transaction which receives a `retry` by another committing transaction. During the `retry`-phase the transaction must not hold locks on `TVars`. In our implementation a transaction that does not get all locks in $T \cup (L_a \setminus L_n)$ releases all locks it already holds and it adds itself to the waiting queue of the `TVar` which was locked by some other thread.

4.4.2 Sending retry-Notifications

The CSHF semantics injects `retry` in conflicting threads, i.e. those threads that have registered themselves in the notify lists of `TVars` which are written by the committing transaction. As already explained this injection is implemented in our library by throwing exceptions. Indeed `sendRetryWithLog` uses the following subprocedure to throw the exceptions:

```
notify []       = return ()
notify (tid:xs) = throwTo tid (RetryException) ≫ notify xs
```

If a conflicting transaction has registered itself in more than one notify list of to-be-written `TVars`, then the CSHF-semantics notifies the thread several times. As an optimization our implementation keeps track of notified threads and throws the exception only once for every thread identifier. However, if several transactions commit concurrently, it may still happen that more than one exception is thrown to the same thread. To ensure that several exceptions do not break the state of the `MVars`, we used `uninterruptibleMask` to protect the `retry`-phase against receiving further exceptions.

4.5 Experimental Results

We tested our SHFSTM and also other implementations (i.e. the GHC implementation, and the ones in [HK06, Boi11]) using some test cases of the STM benchmark of [PSS$^+$08] and also two other tests (SM and $\mathsf{SM}_{\mathsf{ack}}$). The tests where executed on a linux machine with an Intel i7-2600 CPU processor, 8GB main memory, and compiled with GHC 7.4.2 and with optimization `-O2`. We performed tests on a single processor without turning `-threaded` on during compilation (denoted by "seq." in the following table), and with `-threaded` on multiple processors (1,2,3 and 4 cores).

Table 1 shows the mean runtime of 15 runs for every of the following tests:

- **SM** This test uses a map of 200 (key,`TVar`)-pairs stored in a `TVar`. Every of 200 threads reads all `TVars` and writes the sum in the last `TVar`.
- **$\mathsf{SM}_{\mathsf{ack}}$** The test is similar to SM (with 40 threads, 5 `TVars`), where all `TVars` are filled with the value 3 and for every lookup $\mathrm{ackermann}(i, 3)$ is computed where i is between 6 and 8, depending on the thread number. The sum of these numbers and the thread number is written into the last `TVar`. Thus the runtime of the concurrent transactions are different, the last `TVar` is filled with different values.
- **SInt** The test runs 200 threads where every thread increases a single shared `TVar` 200 times.
- **LL** For this test every of 200 threads performs 100 insertions and deletions of random data into a linked list of initial length 300 which is built from `TVars`.
- **BT** For this test every of 200 threads performs 100 insertions and deletions of random data into an (unbalanced) binary tree of initial size 300 built from `TVars`.
- **SK** The test solves concurrently Sudoku-Puzzles where the cells are stored in `TVars`.
- **HT** For this test every of 100 threads performs 100 insertions and deletions on a hashtable built from `TVars`.

The results show that GHCSTM performs much better than the other implementations. An exception is the test $\mathsf{SM}_{\mathsf{ack}}$, where SHFSTM has the best performance. The reason may be that SHFSTM restarts conflicting transactions earlier than e.g. the GHC implementation.

STM-library	#cores	SM	SM$_{ack}$	SInt	LL	BT	HT	SK
GHCSTM	seq	0.04	**2.54**	**0.02**	0.16	0.08	**0.03**	0.70
	1	0.04	2.56	0.03	0.17	**0.06**	0.04	0.73
	2	0.03	2.62	0.03	0.13	0.06	0.04	0.37
	3	0.03	8.59	0.04	0.13	0.09	0.06	0.23
	4	**0.03**	30.49	0.04	**0.13**	0.10	0.06	**0.19**
SHFSTM	seq	0.28	2.62	**0.11**	**5.71**	**1.28**	**0.09**	3.35
	1	**0.27**	2.64	0.14	6.23	1.31	0.10	3.58
	2	2.20	1.66	52.58	28.01	4.00	3.10	2.69
	3	1.95	1.39	69.44	27.64	4.34	2.40	2.28
	4	1.88	**1.08**	75.99	29.34	4.26	2.40	**2.23**
[HK06]	seq	**0.19**	2.49	**0.15**	**7.86**	**0.23**	**0.04**	**20.64**
	1	1.25	2.52	0.36	10.81	0.34	0.06	22.67
	2	26.58	3.04	5.13	49.55	0.47	0.33	24.86
	3	25.25	**2.11**	8.97	61.70	0.39	0.45	25.50
	4	29.37	2.37	11.73	68.02	0.31	0.51	32.37
[Boi11]	seq	0.40	25.50	1.88	3.00	**0.43**	0.14	2.70
	1	0.45	25.70	**0.76**	3.18	0.59	**0.12**	2.70
	2	0.24	12.29	6.68	4.90	0.70	1.39	2.60
	3	0.19	8.58	28.95	2.49	6.92	7.88	1.32
	4	**0.11**	**6.74**	23.76	**1.82**	8.95	6.24	**1.07**

Table 1: Runtimes in sec., bold numbers: best per library, gray background: best of all libraries

Another observation is that parallelisation does not lead to a speedup, but to a slowdown, for the three implementations in Haskell and the tests with high contention. A reason may be that on a single processor, there are not so many conflicting transactions, since they more or less run sequentially. Comparing the three Haskell implementations shows that performance depends on the specific problem and on the concrete scheduling of threads.

5 Conclusion

We explained the CSHF-semantics and discussed our implementation of this approach to STM Haskell in Haskell. The results show that our approach works, but considering efficiency the STM-implementation in GHC performs much better. A general conclusion is that STM implementations should take care to stop nonterminating transactions if they become conflicting which is not the case for all STM Haskell implementations. For further work we may optimize our implementation, especially by replacing the maps and sets inside the `TVars` with lock-free data structures like e.g. concurrent skip-lists. Another approach would be to combine the test on the content change (as GHC does) with our approach of notifying conflicting transactions.

Acknowledgements I thank the anonymous reviewers for their valuable comments.

References

[BMT10] A. Bieniusa, A. Middelkoop, and P. Thiemann. Actions in the Twilight: Concurrent irrevocable transactions and Inconsistency repair. Technical Report 257, Institut für Informatik, Universität Freiburg, 2010.
http://proglang.informatik.uni-freiburg.de/projects/syncstm/techreport2010twilight.pdf.

[Boi11] A. R. Du Bois. An Implementation of Composable Memory Transactions in Haskell. In S. Apel and E. K. Jackson, editors, *SC'11*, *LNCS* 6708, pages 34–50. Springer, 2011.

[DSS06] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In S. Dolev, editor, *DISC'06*, *LNCS* 4167, pages 194–208. Springer, 2006.

[GK08] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPoPP'08*, pages 175–184. ACM, 2008.

[HK06] F. Huch and F. Kupke. A High-Level Implementation of Composable Memory Transactions in Concurrent Haskell. In A. Butterfield, C. Grelck, and F. Huch, editors, *IFL'05, Revised Selected Papers*, *LNCS* 4015, pages 124–141, 2006.

[HMPH05] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In *PPoPP'05*, pages 48–60, New York, NY, USA, 2005. ACM.

[HMPH08] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. *Commun. ACM*, 51(8):91–100, 2008.

[PGF96] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *POPL'96*, pages 295–308. ACM, 1996.

[PSS$^+$08] C. Perfumo, N. Sönmez, S. Stipic, O. S. Unsal, A. Cristal, T. Harris, and M. Valero. The limits of software transactional memory (STM): dissecting Haskell STM applications on a many-core environment. In A. Ramírez, G. Bilardi, and M. Gschwind, editors, *CF'08*, pages 67–78. ACM, 2008.

[Sab12] D. Sabel. An Abstract Machine for Concurrent Haskell with Futures. In S. Jähnichen, B. Rumpe, and H. Schlingloff, editors, *ATPS'12, SE'12 Workshopband*, *GI LNI* 199, pages 29–44, 2012.

[SSS11] D. Sabel and M. Schmidt-Schauß. A contextual semantics for Concurrent Haskell with futures. In *PPDP'11*, pages 101–112, New York, NY, USA, 2011. ACM.

[SSS12] D. Sabel and M. Schmidt-Schauß. Conservative Concurrency in Haskell. In N. Dershowitz, editor, *LICS'12*, pages 561–570. IEEE, 2012.

[SSS13] M. Schmidt-Schauß and D. Sabel. Correctness of an STM Haskell implementation. In G. Morrisett and T. Uustalu, editors, *ICFP'13*, pages 161–172, New York, NY, USA, 2013. ACM.

[ST95] N. Shavit and D. Touitou. Software Transactional Memory. In *PODC'95*, pages 204–213, 1995.

[ST97] N. Shavit and D. Touitou. Software Transactional Memory. *Distributed Computing, Special Issue*, 10:99–116, 1997.