A Rule-Based Specification of Software Transactional Memory

Martin Sulzmann[1] and Duc Hiep Chu[2]

[1] Programming, Logics and Semantics Group, IT University of Copenhagen
Rued Langgaards Vej 7, 2300 Copenhagen S Denmark
martin.sulzmann@gmail.com
[2] School of Computing, National University of Singapore
S16 Level 5, 3 Science Drive 2, Singapore 117543
u0407004@nus.edu.sg

Abstract. Software Transactional Memory (STM) has the promise to avoid the common pitfalls of locks when writing thread-based concurrent programs. Many papers on the subject deal with low-level implementation details to support the efficient and concurrent execution of multiple transactions. We give a rule-based specification of Software Transactional Memory in terms of Constraint Handling Rules (CHR) which naturally supports the concurrent execution of transactions. Such a high-level description of STM in terms of CHR has the advantage that we can easier understand the workings of STM and we can better analyze and verify STM. We verify correctness of a particular CHR-based STM implementation.

1 Introduction

Given the current trend towards multi-core processor architectures, programmers will have to write concurrent programs in order to obtain significant performance improvements. There are numerous approaches for writing concurrent programs such as threads, message-passing etc. We consider here a thread-based model where traditionally locks have been used to avoid data races. The recently popular becoming concept of Software Transactional Memory (STM) has the promise to avoid the common pitfalls of locks (e.g. releasing/acquiring locks too early/soon etc).

A STM transaction is a series of reads and writes to shared memory. Atomic execution of a transaction guarantees that these reads and writes logically either occur all at once, thus, ensuring that intermediate states are not visible to other transactions, or they happen not at all, for example in case two atomic transaction make conflicting updates. The STM run-time guarantees that in case of conflicts at least one transaction can successfully commit its updates whereas the other transaction is retried. In short, concurrent programming in STM is "easy". We simply declare a program region to be executed atomically. That's it!

STM is provided by a number of programming languages, either directly supported by the compiler [6] or provided as a library [7]. Much effort has been

```
newTVar   :: a -> STM (TVar a)
readTVar  :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
atomically:: STM a -> IO a
retry     :: STM ()
forkIO    :: IO a -> IO ThreadId
```

Fig. 1. STM Operations in GHC Haskell

spent so far on efficient implementations of STM. See [14] for numerous references. The literature on high-level STM descriptions (for example see [1, 5, 8, 9, 11, 12]) is comparatively small.

A high-level description of STM has a number of advantages. For example, we can better explain the inner workings of STM and the numerous alternative approaches how to detect and resolve conflicts. A formally specified and verified high-level STM description is a first step to obtain a fully verified STM implementation.

In this paper, we give a rule-based specification of STM described in terms of Constraint Handling Rules (CHR) [3] which is a declarative language to specify concurrent rewritings among multi-sets of constraints. To the best of our knowledge, a rule-based specification of STM appears to be novel. In our view, the advantage of concurrent CHR rewritings are that they naturally model concurrent threads operating on shared data structures. The STM manager is implemented by imposing stricter rules on the set of allowable read/write operations.

Specifically, we make the following contributions:

– We specify Software Transactional Memory via Constraint Handling Rules (Section 4).
– We show that our CHR-based implementation of STM is correct by verifying important properties such as atomicity and isolation. (Section 5).

The next section gives an introduction to STM. Section 3 shows how to represent shared memory operations in CHR. Related work is discussed in Section 6. We conclude in Section 7.

2 STM Examples

We illustrate programming in STM using the GHC Haskell compiler (GHC) [4]. Haskell is a strongly typed language where (monadic) types are used to distinguish among the different kinds of computations/effects. Hence, we can understand the computational behavior (effects) of a Haskell function by simply looking at its type.

STM computations are simply another form of a effect-full computation. The type (constructor) `STM` represents an STM computation and the (abstract) type `TVar a` represents a pointer to a (shared) memory location which holds a value of type `a`. The interface to the STM library in GHC is summarized in Figure 1.

Thus, we can write a program to increment a transactional variable.

```
incSTM :: TVar Int -> STM ()
incSTM x = do { v <- readTVar x ;   -- (1)
                writeTVar x (v+1) }  -- (2)
```

We use here the "`do`" notation to glue together little side-effecting programs to make bigger side-effecting programs. Via `v <- readTVar x` we first read the content of the transactional variable `x` before writing `v+1` into the variable. The whole point of STM is to guarantee that memory transactions are executed atomically. For example,

```
incAtomic :: TVar Int -> IO ()
incAtomic x = atomically (incSTM x)
```

Atomic execution gives us the guarantee that if the value `x` is pointing to has changed after the read operation at location `(1)`, we will not commit the write operation at location `(2)`. Instead, we re-run the entire `incSTM` transaction and only commit if the initially read value remains unchanged during the entire life time of the transaction (that is, after execution of the write operation).

Why does atomic execution yield a IO effect? The Haskell type system disallows to mix programs using different kind of effects. That is, within an STM computation there can only be STM operations but there cannot be any other, say, IO operations. This is a big plus of Haskell's STM because we definitely do not want to re-run IO operations such as "fire missile". However, ultimately we want to run the transaction and make its return result available to other program parts. Therefore, we (atomically) turn the STM computation into a IO computation.

Here is a more realistic (and classic) example of an atomic bank transfer.

```
transfer :: TVar Int -> TVar Int -> Int -> STM ()
transfer fromAcc toAcc amount =
  do { f <- readTVar fromAcc
     ; if f < amount then retry
        else do { writeTVar fromAcc (f-amount)
                ; t <- readTVar toAcc
                ; writeTVar toAcc (t+amount) } }
```

We want to transfer `amount` currency units from `fromAcc` to `toAcc`. If the balance of `fromAcc` is insufficient we simply retry. That is, we abort the transaction and try again. There is no point in re-running the transaction if `fromAcc` has not changed. Hence, the transaction simply blocks until `fromAcc` has been updated. Let's see what happens in case there are concurrent transfers involving the same set of accounts.

```
multiple_transfers =
  do { a1 <- atomically (newTVar 100)
     ; a2 <- atomically (newTVar 50)
     ; forkIO (atomically (transfer a1 a2 40))    -- t1
     ; forkIO (atomically (transfer a2 a1 60)) }  -- t2
```

We concurrently transfer `40` currency units from `a1` to `a2` and `60` currency units from `a2` to `a1` by forking two threads. Both transfers are executed atomically which gives us the guarantee that the transfer either happens all at once or not at all and none of the intermediate (transfer) steps are observable. If transfer `t2` is executed first, we block because the funds in account `a2` are insufficient. Once transfer `t1` is executed, transfer `t2` is re-started. As a final result, `a1` holds `120` currency units and `a2` holds `30` currency units. Execution of `t1` before `t2` leads to the same result. The behavior of the program is deterministic (which is of course not always the case for a concurrent program).

3 Shared Memory Operations and CHR

Our first task is to model shared memory and its associated read and write operations in CHR. We will show how to enforce atomicity and isolation of read/write operations in the next section.

CHR manipulate a global constraint store in terms of (left-to-right) rewrite rules. Rewrite rules can be applied concurrently as long as their left-hand sides do not overlap. We represent shared memory locations and read/write operations via the following CHR constraints

```
Cell(l,v)    cell at location l storing value v
Read(l,v)    reads value v from location l
Write(l,v)   writes v to location l
```

The interaction between read/write operations and shared memory is specified via the following CHR rules.

```
read  @ Cell(l,v1) \ Read(l,v2) <=> v1 = v2
write @ Cell(l,v1), Write(l,v2) <=> Cell(l,v2)
```

For example, the first rule rewrites `Read(l,v2)` to `v1 = v2` if there is `Cell(l,v1)` in the store. That is, we read from a cell by binding variable `v2` to the value `v1` stored in the memory cell at location `l`. The point to note is that not the entire left-hand side is rewritten to the right-hand side. We only rewrite the part after the \ but the part before \ can be shared among other concurrent reads. In the following variant

```
read'  @ Cell(l,v1), Read(l,v2) <=> Cell(l,v1), v1 = v2
```

`Cell(l,v1)` is removed and immediately added to the store again. But this rule disallows concurrent reads which are possible with `read`. The `write` rule clearly requires exclusive access to the memory cell. We rewrite the entire left-hand side to the right-hand side.

The abstract CHR semantics [3] gives the implementor the freedom to apply rules concurrently and in random order. However, we wish that read/write operations belonging to the same thread of execution are processed in a fixed sequence (following the control flow of the program). Therefore, we adopt the CHR execution model known as the refined operational CHR semantics [2]. We assume that memory cells are kept in the global store but read/write operations are kept on an execution stack. In order to execute a CHR rule, we pop a constraint from the stack and go through the list of rules (from top to bottom) until we find a rule which partially matches (i.e. the popped constraint forms a partial match). Then, we try to complete the match by finding the remaining left-hand side constraints in the store. Right-hand side constraints will either be added to the store or pushed onto the execution stack. We classify constraints into *executable* (for example read/write) and *store* (for example cell) constraints to avoid ambiguities which constraint shall be pushed or added.

We assume that for each CHR rule the left-hand side consists of exactly one *executable* constraint and an arbitrary (possibly zero) number of *store* constraints. The right-hand side consists of at most one *executable* constraint and an arbitrary (possibly zero) number of *store* constraints. This guarantees a deterministic execution strategy written $St \mid E \rightarrowtail_P St' \mid E'$ where St and E are the initial store and execution stack and St' and E' are the resulting store and execution stack after at most one rule from the set P of CHR rules by finding the first match trying the rules from top to bottom. In case none of the rules is applicable we return St and E unchanged.

Concurrent execution of n execution stacks can be described straightforwardly

$$\frac{\begin{array}{c} St \mid E_i \rightarrowtail_P St' \mid E'_i \quad \text{for some } i \in \{1, \ldots, n\} \\ E'_j = E_j \text{ for all } j \in \{1, \ldots, n\} - \{i\} \end{array}}{(St \mid E_1, \ldots, E_n) \rightarrowtail_P (St' \mid E'_1, \ldots, E'_n)}$$

We arbitrarily chose one execution stack (ignoring fairness issues for brevity).

4 STM implemented in CHR

We refine the scheme outlined in the previous section to guarantee STM-style atomic, transactional execution of a sequence of executables. We specify the STM manager via CHR rules.

The STM run-time must guarantee that all reads and writes within a transaction happen logically at once. We can easily achieve this via a "stop-the-world" semantics. That is, at any point there is at most one transaction active. Obviously, this is not efficient because we would like to maximize the amount of parallelism by running as many as possible transactions in parallel, typically, one transaction per available processor core. In case transactions are (optimistically) executed in parallel the STM run-time must take care of any potential read/write conflicts. The idea is to use for each transaction a read and a write log.

The STM run-time records the initial value of each shared pointer in a read log when reading the shared pointer for the first time. When writing to a shared pointer we do not immediately perform the update, rather we record the to-be-written value in a write log. Before we can commit the write log (i.e. actually updating the shared pointers), we first must validate that for each shared pointer whose value is stored in the read log, the actual value stored in the memory cell and in the read log are still the same.

This suggests that we require the CHR constraints mentioned in Figure 2. We assume that the value parameter of a `Cell` constraint is functionally defined by the location and values of `RLog` and `WLog` constraints are functionally defined by the transaction and location. Read and write log constraints are usually local to each transaction but we keep them in the global constraint store (because the STM run-time might want to observe the logs of two transactions).

There is obviously quite a bit of design space how to maintain the read and write log and how to actually perform the validate and commit (for example are parallel validates/commits allowed?). We implement STM using optimistic reads and writes where after executing all operations all reads are validated and if validation is successful all writes are committed. We allow for concurrent validations but there is at most one committer (represented by a single occurrence of `CommitRight` in the initial store). Any validation process running concurrently to a commit process is forced to rollback. The rules implementing this scheme are given in Figure 2.

A description of the rules is given below. The reader should keep in mind that for each execution stack the rules are executed from top to bottom.

- Start rules `s1-3`. We clear the read and write log (in case there is some "noise" from a previous run).
- Read rules `r1-3`: We first check the write/read log. In case of a first initial read, we consult the memory cell and create a read log.
- Write rules `w1-2`. We first check if a write log already exists and update the write log. Otherwise, we create a write log.
- Retry rule `rt`. We rollback if any of the variables in the read log has changed. The execution of a rollback is dealt with by the following rule application step.

$$\frac{\begin{array}{c} E_i = [\texttt{Rollback}(t)|\_] \text{for some } i \in \{1,\ldots,n\} \\ E'_i = E_{initial_i} \quad E'_j = E_j \text{ for all } j \in \{1,\ldots,n\} - \{i\} \end{array}}{(St \mathbin{|} E_1,\ldots,E_n) \rightarrowtail_P (St' \mathbin{|} E'_1,\ldots,E'_n)}$$

  We use Prolog style lists $[x|xs]$ to represent an execution stack with top element x and remaining stack elements xs. $E_{initial_i}$ refers to the initial configuration of the ith stack.
- End rule `e` calls validation, but only if there is no committer working. The store constraint `ValidateOn(t)` allows a committer to rollback a validating transaction.
- Validation rules `v1-3` test for any conflicts among the read log and actual values stored in a cell. If there are none we commit. In order to commit we must require the `CommitRight` constraint. We assume there exists only one of such constraints in the initial store. Hence, there can be at most one committer. Rule `v4` applies if we have no commit right and/or a committer removed the validation right (see rule `c1`).
- The commit rule `c1` forces any concurrent validation to rollback. Rule `c2` commits any write updates. Once all writes are committed we give back the `CommitRight` constraint via rule `c3`.

Store constraints:

```
Cell(l,v)        cell at location l storing value v
RLog(t,l,v)      read log of t, v the initially read value
WLog(t,l,v)      write log of t, v the last written value
ValidateOn(t)    signals that t is validating
CommitRight      has right to commit
```

Executables:

```
Start(t)        t starts
Read(t,l,v)     t reads value v from location l
Write(t,l,v)    t writes v to location l
Retry(t)        t retries
End(t)          t ends => validate then commit
```

Internal executables: `Rollback(t)`, `Validate(t)`, `Commit(t)`
Rules:

```
s1 @ Start(t) \ WLog(t,_,_) <=> True
s2 @ Start(t) \ Rlog(t,_,_) <=> True
s3 @ Start(t) <=> True
r1 @ WLog(t,l,v1) \ Read(t,l,v2) <=> v1 = v2
r2 @ RLog(t,l,v1) \ Read(t,l,v2) <=> v1 = v2
r3 @ Cell(l,v1) \ Read(t,l,v2) <=> v1 = v2, RLog(t,l,v1)
w1 @ WLog(t,l,v1), Write(t,l,v2) <=> WLog(t,l,v2)
w2 @ Write(t,l,v) <=> WLog(t,l,v)
rt @ Cell(l,v1), RLog(t,l,v2) \ Retry(t) <==> v1 =!= v2 | Rollback(t)
e @ CommitRight \ End(t) <=> Validate(t), ValidateOn(t)
v1 @ Cell(l,v1), ValidateOn(t), Validate(t) \ RLog(t,l,v2)
     <=> v1 = v2 | True
v2 @ Cell(l,v1) \ ValidateOn(t), Validate(t), RLog(t,l,v2)
     <=> v1 =!= v2 | Rollback(t)
v3 @ CommitRight, ValidateOn(t), Validate(t) <=> Commit(t)
v4 @ Validate(t) <=> Rollback(t)
c1 @ Commit(t1) \ ValidateOn(t2) <=> True
c2 @ Commit(t) \ Cell(l,v1), WLog(t,l,v2) <=> Cell(l,v2)
c3 @ Commit(t) <=> CommitRight
```

Fig. 2. CHR-Based STM Implementation

5 Soundness of the CHR-Based STM Implementation

For simplicity, we omit the treatment of `retry` operations. We believe that most of the upcoming results (atomicity, isolation correctness) straightforwardly extend to `retry` operations with the exception of optimistic concurrency where we see the potential problem that two retrying transactions (maybe indefinitely) wait for each other.

We formalize some basic assumptions about execution stacks and states.

Definition 1 (Well-Defined Execution Stack and State). *The content of the initial execution stack for a transaction is well defined iff*

- *It starts with a `Start` constraint and ends with an `End` constraint, and*
- *Other operations are either `Read` or `Write` constraints*

The content of the intermediate execution stack for a transaction during execution is well defined iff

- *It is empty, i.e. end of evaluation, or*
- *It is a suffix of the Initial Execution Stack, or*
- *It contains only one constraint, which either is a `Rollback` or `Validate` or `Commit` constraint*

A program state $(St \mid E_1, \ldots, E_n)$ is said to be well-defined iff

- $\forall i \; E_i$ *is well defined*
- $\mathtt{WLog}(\_, l, \_) \in St \vee \mathtt{RLog}(\_, l, \_) \in St \Rightarrow \mathtt{Cell}(l, \_) \in St$
- $\forall i \; \mathtt{Write}(\_, l, \_) \in E_i \vee \mathtt{Read}(\_, l, \_) \in E_i \Rightarrow \mathtt{Cell}(l, \_) \in St$

It is straightforward to verify that if we start in a well-defined state we only reach well-defined states.

In our first result, we establish atomicity. To avoid (indefinitely) stuck transactions, we assume a round-robin scheduler where each transaction/execution stack will be executed in a round-robin fashion.

Definition 2 (Atomicity). *Atomicity means that a transaction either successfully commits or has to rollback.*

Atomicity follows from the following auxiliary lemmas.

Lemma 1. *In any possible interleaving execution, each transaction will reach a state where its execution stack is left with only an `End` constraint.*

Proof. (Sketch) This follows from the fact that a transaction executes `Start`, `Read`, and `Write` constraints on its own without any interference from other transactions.

A transaction might have to wait at the `End` constraint until `CommitRight` is available before we can proceed with validation. Under a round-robin scheduling policy and based on the upcoming lemma each transaction will eventually be able to start validation. During validation, a transaction might be forced to rollback if some other transaction has already progressed to the commit stage or has already successfully committed and updated conflicting values.

Lemma 2. *A transaction, which successfully validates, will eventually commit successfully and thus release/add `CommitRight` to the store which then allows other transactions to start validation.*

Proof. (Sketch) Rule `c1` could not be fired infinitely as rule `e` prevents any transaction from entering validation stage when `CommitRight` has already been acquired. Therefore, committing transaction will always make progress and eventually finish execution.

In summary, under a round-robin scheduler each transaction is always able to start validating. Then, the transaction will either be forced to rollback or successfully validates and commits.

Theorem 1 (Atomicity of CHR-Based STM). *Our implementation guarantees atomicity.*

Definition 3 (Isolation). *Isolation means that the execution of transactions is serializable.*

Theorem 2 (Isolation of CHR-Based STM). *Our implementation guarantees isolation.*

Proof. (Sketch) W.l.o.g., we only consider the case of two active transactions. For this case, isolation means the following:

Suppose $St \shortmid E_1, E_2 \rightarrowtail_P^* St' \shortmid [], []$. Then, either (1) $St \shortmid E_1 \rightarrowtail_P^* St'' \shortmid []$ and $St'' \shortmid E_2 \rightarrowtail_P^* St' \shortmid []$, or (2) $St \shortmid E_2 \rightarrowtail_P^* St''' \shortmid []$ and $St''' \shortmid E_1 \rightarrowtail_P^* St' \shortmid []$ where St'' and St''' are some intermediate stores.

$St \shortmid E_1, E_2 \rightarrowtail_P^* St' \shortmid [], []$ implies that both transactions successfully commit. W.l.o.g., we assume that transaction 1 committed earlier than transaction 2, case (1). Hence, we can assume that $St \shortmid E_1, E_2 \rightarrowtail_P^* St'' \shortmid [], [\texttt{End}(t_2)] \rightarrowtail_P^* St' \shortmid [], []$. The intermediate step represents the situation where transaction 1 has already committed and transaction 2 is just about to validate.

We proceed by considering the following two cases. In $St'' \shortmid [], [\texttt{End}(t_2)] \rightarrowtail_P^* St' \shortmid [], []$, validation of transaction 2 either leads to a rollback or there is no rollback.

– Rollback: This implies that a conflict took place. Hence, we had to restart the entire transaction 2 in the derivation $St'' \shortmid [], [\texttt{End}(t_2)] \rightarrowtail_P^* St' \shortmid [], []$. Hence, we can easily verify that the serialized execution of transaction 1 and then transaction 2 leads to the same result St'. That is, $St \shortmid E_1 \rightarrowtail_P^* St''' \shortmid []$ and $St''' \shortmid E_2 \rightarrowtail_P^* St' \shortmid []$ where St''' is derived from St'' by discarding all of transaction 2's read and write logs.

– No rollback: This implies that transaction 1 and transaction 2 have no (data) conflicts. Hence, their operations are non-interfering. In terms of CHR, their rule applications are joinable (lead to common states). Recall that we assume the refined operational CHR semantics [2] which enforces deterministic, sequential rule execution (per stack/transaction). Hence, we can execute transaction 1 before executing transaction 2 which then leads to the same result St'.

Definition 4 (Optimistic Concurrency of CHR-Based STM). *Optimistic Concurrency means that at least one transaction commits.*

Theorem 3. *Our implementation guarantees optimistic concurrency.*

Proof. The proof proceeds by contradiction. Assume that no transaction is able to commit. Suppose transaction i which can not commit. Under a round-robin scheduler this implies that transaction i is (eventually) forced to rollback. There are only two possible cases: (1) Some transaction has successfully updated values seen by transaction i, or (2) some transaction has acquired `CommitRight` while transaction i is validating.

In both cases, there is a transaction that at least has already proceeded to the commit stage. By Lemma 2, that transaction successfully commits. Contradiction.

Theorem 4 (Correctness of CHR-Based STM). *If a transaction commits successfully, the store reflects correctly all the reads/writes performed by that transaction.*

Proof. This follows from two facts. Our read/write rules ensure that the `RLog` and `WLog` constraints record all effects of the read/write operations. The commit rules ensure that changes are reflected in `Cell` constraints.

6 Related Work

Previous work [1, 5, 8, 9, 11, 12] on high-level STM descriptions has similar goals (i.e. to verify correctness of STM implementations) but differs significanly from our work in terms of the formalism used. For example, the work in [11] uses operational semantics and [8] uses Haskell as a meta specification for describing the behavior of STM. To the best of our knowledge, we are the first to give a rule-based specification of STM, Concretly, we use CHR to specify STM.

Interestingly, in our own previous work [13] we have employed STM to efficiently implement CHR. In this light, the present work shows that both concurreny models, STM and CHR, are equally expressive and can be used to encode each other.

7 Conclusion and Future Work

We have formalized a particular form of STM in terms of CHR. Our STM implements a lazy conflict detection scheme which allows for parallel validation of read sets but only supports at most one committer at a time. We could verify important properties such as atomicity and isolation. We have also implemented the described STM in terms of a Haskell-CHR library and will release the source code in the near future (via the Haskell platform 'hackage').

In future work, we plan to consider further variants of STM and their CHR-based formulation. For example, the high-level CHR description of STM makes it possible to customize STM to domain-specific settings. An interesting aspect is how/whether we can compose different STM formulations, i.e. CHR rule sets, while maintaining atomicity and isolation.

In another direction, we want to exploit the close connection between CHR and concurrent logic frameworks [10] to mechanically verify our so far hand-written proofs.

Acknowledgments

We thank the LOPSTR'08 reviewers of the pre-proceedings round for their feedback.

References

1. A. Cohen, J. W. O'Leary, A. Pnueli, M. R. Tuttle, and L. D. Zuck. Verifying correctness of transactional memories. In *Proc. of FMCAD'07*, pages 37–44. IEEE Computer Society, 2007.
2. G. J. Duck, P. J. Stuckey, M. J. García de la Banda, and C. Holzbaur. The refined operational semantics of Constraint Handling Rules. In *Proc of ICLP'04*, volume 3132 of *LNCS*, pages 90–104. Springer-Verlag, 2004.
3. T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, 37(1-3):95–138, 1998.
4. Glasgow haskell compiler home page. http://www.haskell.org/ghc/.
5. R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proc. of PPOPP'08*, pages 175–184. ACM Press, 2008.
6. T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In *Proc. of PPoPP'05*, pages 48–60. ACM Press, 2005.
7. M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. *SIGPLAN Not.*, 41(10):253–262, 2006.
8. L. Hu and G. Hutton. Implementing software transactional memory, correctly. Presented at TFP'2008.
9. F. Huch and F. Kupke. A high-level implementation of composable memory transactions in Concurrent Haskell. In *Proc. of IFL'05*, volume 4015 of *LNCS*, pages 124–141. Springer-Verlag, 2005.
10. P. López, F. Pfenning, J. Polakow, and K. Watkins. Monadic Concurrent Linear Logic Programming. In *Proc. of PPDP'05*, pages 35–46, 2005.
11. K. F. Moore and D. Grossman. High-level small-step operational semantics for transactions. In *Proc. of POPL'08*, pages 51–62. ACM Press, 2008.
12. M. L. Scott. Sequential specification of transactional memory semantics, 2006. Proc. of TRANSACT'06: First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing.
13. M. Sulzmann and E. S. L. Lam. Parallel execution of multi set constraint rewrite rules. In *Proc. of PPDP'08*, 2008.
14. Transactional memory bibliography. http://www.cs.wisc.edu/trans-memory/biblio/index.html.