A Property Based Testing Approach for Software Transactional Memory Safety

Anonymous Author(s)

Abstract

Software Transactional Memory (STM) provides programmers with a simple high-level model of transactions that allows the writing of concurrent programs without worrying with locks, since all transaction concurrency management is done by the STM runtime. Such programming model greatly simplifies development of concurrent applications, but it has a cost: implementing an efficient and correct STM algorithm is an art. Several criteria have been proposed to certify STM algorithms, some based on model checkers and proof assistants. Such approaches are heavyweight and do not allow quick experimentation with different TM algorithm designs. In this work, we propose a more lightweight approach: specify STM algorithm as small-step operational semantics of a idealized language with STM support and check for safety properties using QuickCheck, a property-based testing library for Haskell.

CCS Concepts •**Computer systems organization** → **Embedded systems;** *Redundancy;* Robotics; •**Networks** → Network reliability;

Keywords Software Transactional Memory, Semantics, Opacity, Markability, Property-based Testing

ACM Reference format:
Anonymous Author(s). 2017. A Property Based Testing Approach for Software Transactional Memory Safety. In *Proceedings of ACM Haskell Symposium, El Paso, Texas USA, July 2017 (HASKELL'17),* 12 pages.
DOI: 10.475/123_4

1 Introduction

Transactional Memory (TM) [20, 38] provides programmers a high level concurrency control abstraction. Programmers can simply declare certain code pieces as transactions and the TM runtime guarantees that transactions execute in isolation. The use of TM provides atomicity, deadlock freedom, composability [17] and increases productivity when compared to using locks [37]. Several works developed software [8, 18, 19], hardware [14] and software / hardware hybrids [2, 6] implementations. Gradually, industry is adopting TM: IBM Blue Gene/Q processor supports TM and Intel Haswell microarchitecture supports transaction synchronization primitives [15, 25].

The TM runtime is responsible to ensure correct management of shared state. Therefore, correctness of TM clients depend on a correct implementation of TM algorithms. However, this simple programming model has a price: designing a correct TM algorithm is an art. Researchers use different techniques to implement the TM interface efficiently. Algorithms try to interleave transactions as much as possible, while guaranteeing a non-interleaving semantics. Thus, subtle but fast algorithms are favored over simpler ones and such subtlety makes them prone to intricate bugs.

A first step towards correct implementation of TM algorithms is a specification of what is correctness for TM. Intuitively, a correct TM algorithm should guarantee that every execution of an arbitrary set of transactions is indistinguishable from a sequential running of them. Several correctness criteria where proposed in the literature [9, 12, 24, 31] and they rely on the concept of transactional histories. Intuitively, a history consists of a sequence of operations executed on shared objects during a TM execution. Analysing TM history structure generated by algorithms, we can ensure that its TM interface provides atomicity and deadlock freedom to client applications. However, certify that a TM algorithm is safe according to some criteria is a non-trivial task. Different works use I/O automata [29], model checking [4, 5, 11] or define a specification language that reduces the problem of proving non-opacity of a TM algorithm to SMT solving [7, 30].

Such correctness concerns are not just formalization curiosity, they can influence directly on implementation efficiency. Le et.al. [28] mention that current STM-Haskell implementation does not enjoy opacity and that it can cause threads to loop, due to accessing an inconsistent memory view. To avoid such problems STM-Haskell implementation validates the read set each time a thread is scheduled and such checking overhead can cause a waste of execution time. This is one of the motivation for Le et. al. [28] proposing a new implementation of STM-Haskell using the Transaction Locking II (TL2) algorithm [8].

Semanticists have devoted attention to formally specify TM behavior [1, 17, 26, 33]. Harris et. al. specifies STM support to Haskell programming language as an extension of GHC compiler [17], defines a semantics for a core functional language with STM features and they argue that their semantics is "small-step", but when evaluating an orElse operation or a atomically block, the semantics uses "multi-steps" to evaluate such constructs using the reflexive-transitive closure of single step semantics. Moore et. al. specifies a small-step semantic for TM providing textual proofs of some safety properties, but their semantic does not provide support for high-level transactional primitives like orElse and retry [33]. Hu et. al. uses property based testing for checking a compiler from a high level STM language to a transactional virtual machine by defining a stop-the-world semantics for the high-level language and a semantics for the virtual machine [21]. Hu et. al. high-level semantics also uses multi-steps to evaluate transactions and does not provide support for high-level operators like orElse and retry.

Defining formal semantics and correctness criteria are fundamental steps to ensure safety of TM algorithms. To the best of our knowledge, there is no truly small-step semantics for TM such that: 1) consider high-level constructs like orElse and retry while allowing the interleaving of executing transactions and 2) produces a history of TM execution that can be used to verify safety of TM semantics. This work aims to fill this gap. Our approach is to specify

HASKELL'17, El Paso, Texas USA

DOI: 10.475/123_4

STM algorithms using a standard small-step operational semantics for a simple transactional language and use property based testing to check if safety properties are satisified by histories generated. We are aware that using automated testing isn't sufficient to ensure correctness of an algorithm, but it can expose bugs before using more formal approaches, like formalizing the algorithm in a proof assistant.

Specifically, we made the following contributions:

- We define a simplified model language that supports high-level TM constructs orElse and retry present in STM-Haskell.
- We define two trace based small-step operational semantics for a high-level language with STM support. One semantics closely follows the well-known TL2 algorithm [8] and the other is based on the semantics adopted by STM Haskell [17]. These semantics are implemented in the Haskell programming language.
- We define TM safety conditions, namely opacity and markability [9, 12, 24, 31], in Haskell and check them using QuickCheck [3] against the implemented semantics. Defining safety properties is just a matter to define functions that verify them on histories produced by interpreters implementing TM algorithm semantics.
- We check that a compiler from our high-level language to a transactional stack-machine preserves safety properties. To check this we need to define a trace based semantics for the stack-machine in order to record transactional histories and to fully support our simplified high-level language.

The rest of this paper is organized as follows. Section 2 presents a brief introduction to software transaction memory, specially its implementation in the Haskell programming language. Section 3 defines the syntax (Section 3.1) and operational semantics based on TL2 and on STM-Haskell (Section 3.2) for a small STM-Haskell like language. Section 4 presents an extension of the virtual machine and compiler from the high-level language to virtual machine. In Section 5 we present Haskell implementations of TM safety properties and describe how to check them using QuickCheck, giving some details on how random test cases are generated and describe test coverage results. Section 7 discuss related work and Section 8 concludes and presents future work.

2 Software Transaction Memory

STM design space: In an STM system, memory transactions can execute concurrently and, if finished without conflicts, a transaction may commit. Conflict detection may be *eager*, if a conflict is detected the first time a transaction accesses a value, or *lazy* when it occurs only at commit time. With eager conflict detection, to access a value, a transaction must acquire ownership of the value, hence preventing other transactions to access it, which is also called pessimistic concurrency control. With optimistic concurrency control, ownership acquisition and validation occurs only when committing. These design options can be combined for different kinds of accesses to data, e.g., eager conflict detection for write operations and lazy for reads. STM systems also differ in the granularity of conflict detection, word based and object based being the most common. STM systems need a mechanism for version management. With eager version management, values are updated directly in memory and a transaction must maintain an undo log where it keeps the original values. If a transaction aborts, it uses the undo log to copy the old values back to memory. With lazy version management, all writes are buffered in a redo log, and reads must consult this log to see earlier writes. If a transaction commits, it copies these values to memory, and if it aborts the redo log can be discarded. An STM implementation can be lock based, or obstruction free. An obstruction free STM does not use blocking mechanisms for synchronization and guarantees that a transaction will progress even if all other transactions are suspended. Lock based implementations, although offering weaker progress guarantees, are believed to be faster and easier to implement.

TL2 is a classic lock based, deferred update transactional algorithm, similar to the one used in the GHC implementation of STM Haskell [17]: all writes are recorded in a write-set. When a transaction finishes, it validates its log to check if it has seen a consistent view of memory, and its changes are committed to memory. The main difference of the TL2 algorithm is that conflicts are detected by using a global clock that is shared by all threads. When a transaction starts executing it reads the global clock to get its read stamp. Every transacted memory location is associated with a write stamp, when opening an object for reading/writing, the transaction checks if the write stamp of memory address is not greater than the transaction read stamp, in that case it means that the object was modified after the transaction started executing, hence the transaction must be aborted. If the memory passes the validation it means that the transaction has seen a consistent view of the memory.

STM Haskell: Harris et.al. [17] extends Haskell with a set of primitives for writing memory transactions. The main abstractions are transactional variables or TVars, which are special variables that can only be accessed inside transactions. Below, the main STM Haskell primitives are shown. The readTVar takes a TVar and returns a transactional action STM a. This action, when executed, will return a value of type a, i.e., the TVar's content. Similarly, writeTVar takes a value of type a, a TVar of the same type and returns a STM action that when executed writes into the TVar.

```
data TVar a = ...
data STM a = ...
writeTVar :: TVar a → a → STM ()
readTVar :: TVar a → STM a
retry :: STM ()
orElse :: STM a → STM a → STM a
atomically :: STM a → IO a
```

These transactional actions can be composed together to generate new actions through monadic combinators or the do notation:

```
type Var = TVar Float
transferMoney :: Float → Var → Var → STM ()
transferMoney amount acc1 acc2 = do
    v ← readTVar acc1
  if v ⩾ amount
    then do
      writeTVar acc1 (v − amount)
      v2 ← readTVar acc2
      writeTVar acc2 (v2 + amount)
    else retry
```

The retry primitive is used to abort and re-execute a transaction once at least one of the memory positions it has read is modified and orElse is a choice operator, it takes two transactions as arguments, if the first one retries then the second one is executed. If both fail the whole transaction is executed again. Transactions are executed with atomically:

```
atomically (transferMoney 100.00 tvar1 tvar2)
```

It takes as an argument a transactional action (STM a) and executes it atomically with respect to other concurrently executing transactions.

3 A Model for Software Transactional Memory

Our objective is to formalize semantic that ensure, by construction, that an implementation for transactional memory enjoy safety properties, namely opacity and markability.

3.1 Language Syntax

Our minimalistic language is defined by data types Tran, which represents computations in the STM monad, and Proc that denotes actions in the Haskell IO monad. This language is, in essence, the same as the one proposed by [21]. We extend it with orElse, retry, conditional constructs and a special value to denote a aborted transaction, TAbort. Such constructs aren't an essential part of a model for TM, but they interesting on their own when we consider safety properties of TM.

```
newtype Val = Val {unVal :: Int}

newtype Var = Var {unVar :: Int}

newtype Id = Id {unId :: Int}

data Tran = TVal Val | TRead Var | TWrite Var Tran
 | Tran ⊕T Tran | TIf Tran Tran Tran
 | TOrElse Tran Tran | TRetry | TAbort

data Proc = PVal Val | PFork Proc
 | PAt (Maybe (Id, Stamp)) Tran
 | Proc ⊕P Proc
```

In order to avoid dealing with name binding, we do not provide a language construct for creating new variables and also use addition operation for composing transactions and processes. This is valid simplification, since addition forms a monoid on integer values, while still retaining the sequencing computations and combining their results. We represent variables and values by integers (properly wrapped using a **newtype**). Each syntax construct meaning is immediate with the exception of how we represent atomic blocks. A value built with constructor PAt carries information about its transaction id and current transaction read stamp. Initially, all PAt values are built using Nothing to denote that such block did not started its execution. To avoid clutter in the presentation of PAt semantics, we represent information about transaction id and read stamp as (i, j) whenever it has the form Just (i, j) and as $()$ if it is equal to Nothing. Also, we allow ourselves a bit of informality and write Ids, Stamps, Val and Var values as if they were simple integers.

Construction TAbort is used to represent a transaction that is aborted by accessing an inconsistent view of memory, in our TL2-based semantics, and by trying to commit an transaction that which has accessed and invalid memory configuration, in STM-Haskell based semantics. Term TAbort does not appear on randomly generated source programs. It is used in our semantics to properly differ between different types of transaction aborting and how they should be treated by semantics of orElse construct, since transactions aborted by inconsistent views should not be captured by an orElse.

Example 1. As an example of how our tiny language can model STM constructs, consider the following function that increments an value stored on a TVar, if it is different from zero.

```
incVar :: TVar Int → STM ()
incVar v
  = do
    x ← readTVar v
    if x ≡ 0 then return ()
      else writeTVar v (x + 1)
```

Such function can be represented in our language as follows:

```
incVar v = TIf (TRead v) 0 (TWrite v (TRead v ⊕T 1))
```

3.2 Language Semantics

In this section, we define two operational semantics for our STM language. First, we present a semantics inspired by the TL2 algorithm which unlike previous works [17, 21] uses heaps, transaction logs (read and write sets) and also records the event history of current TM execution. The use of transaction logs on a high-level semantics is a bit unusual, but necessary to proper modeling of commit and abort operations of different TM algorithms. Next, we propose a semantics inspired by STM Haskell in which no global clock is used for TVar version control.

Before presenting the semantics, we need to define some notation. We use finite maps to represent heaps and logs used by transactions (i.e. read and write sets). Notation • denotes an empty finite mapping and Θ represents a 4-uple formed by a heap and mappings between transaction id's and their read / write sets and transactions. We let Θ_h, Θ_r, Θ_w, and Θ_T represent the heap and finite functions between transaction ids and their logs and transactions, respectively. Let $h(x)$ denote the operation of retrieving the value associated with key x in finite mapping h and $h(x) = \bot$ denotes that no value is associated with x. Notation $h \uplus h'$ denotes the right-biased union of two finite mappings, i.e. when both maps have the same key x, we keep the value $h'(x)$. We let $\Theta_r(x, j)$ denote the operation of retrieving the value associated with key x in the read set of transaction with id j. Notation $\Theta_w(x, j)$ is defined similarly for write sets. Updating a variable x with value v in the read set of transaction j is denoted as $\Theta_r\,[j, x \mapsto v]$. Same holds for write set Θ_w. Finally, notation $h\mid_x$ denotes the finite mapping h' with entries for the key x removed, i.e. $h' = h - [x \mapsto v]$, for some value v.

Operations executed on transactional variables during a TM execution are represented by data type Event. Essentially, Event records operations on variables and on transactions. A history of a TM execution is formed by a list of Events.

```
newtype Stamp = Stamp {unStamp :: Int}

data Event = IRead Id Var Val
   | IWrite Id Var Val
   | IBegin Id
   | ICommit Id
```

```
  | IAbort Id
  | IRetry Id
type History = [Event]
```

Data type Stamp denotes the global clock used by the TL2 algorithm to ensure correct variable versions. Constructors IBegin, ICommit and IAbort denote the beginning, commit and failure of a transaction with a given Id. We consider that a transaction fails when it tries to read from an inconsistent view of memory. IRead id v val records that value val was read by transaction id for variable v and IWrite id v val denotes that value val was written in variable v by transaction id. An event IRetry denotes the user called retry on current transaction and such transaction should be restarted. In our semantics, the computation of histories is represented as a Writer monad, which adds a new element by appending at history end. But, for presentation purpouses, we simply add a new event at head of a given history.

3.2.1 TL2 Based Semantics

Now, we present our semantics based on Transactional Locking 2 algorithm [8]. Informally, TL2 algorithm works as follows: threads execute reads and writes to objects, but no memory locations are actually modified. All writes and reads are recorded in write and read logs. When a transaction finishes, it validates its log to check if it has seen a consistent view of memory, and its changes are committed to memory.

Function $\Theta(x, i, j)$ denotes that transaction j with read-stamp i tries to to read the content of variable x and it works as follows: First it checks the write set. If the variable has not been written to, the read set is consulted. Otherwise, if the variable has also not been read, its value is looked up from the heap and the read log updated accordingly, if variable's write stamp is not greater than current transaction read-stamp i. Otherwise, we have an conflict and the current transaction is aborted by returning TAbort.

$$\Theta(x,i,j) = \begin{cases} (v,\Theta) & \mathit{if}\ \Theta_w(x,j) = (i',v) \\ (v,\Theta) & \mathit{if}\ \Theta_w(x,j) = \bot, \Theta_r(x,j) \neq \bot, \\ & \quad \Theta_h(x) = (i',v)\ \mathit{and}\ i \geqslant i' \\ (v,\Theta_r\,[j, x \mapsto v]) & \mathit{if}\ \Theta_w(x) = \Theta_r(x) = \bot, \\ & \quad \Theta_h(x) = (i',v),\ \mathit{and}\ i \geqslant i' \\ \text{TAbort} & \mathit{if}\ \Theta_h(x) = (i',v)\ \mathit{and}\ i < i' \end{cases}$$

Figure 1. Reading a variable

Predicate $consistent(\Theta, i, j)$ holds if transaction j has finished its execution on a valid view of memory. We say that a TM state Θ is consistent if all variables read have stamps less than or equal to i, the global clock value in which transaction j have started.

$$consistent(\Theta,i,j) = \forall x.\Theta_r(x,j) = (i,v) \rightarrow \Theta_h(x) = (i',v) \land i \geqslant i'$$

Figure 2. Predicate for consistency of transaction logs

In order to provide a concise semantics definition, in Figure 3, we define evaluation contexts to avoid the need of "congruence rules". In our semantics definition we use the following meta-variable convention: we let t denote arbitrary transactions and p processes. Values are represented by v, stamps by i and transaction ids by j. All meta-variables can appear primed or subscripted, as usual. Finally, in order to avoid several rules to propagating different types of transaction failure, we use TFail whenever any of TAbort or TRetry applies. Same holds for events: IFail will represent any of IAbort or IRetry.

Evaluation contexts for transactions

$$\begin{array}{lcl} \mathbb{T}[\cdot] & ::= & \text{TWrite}\ v\ \mathbb{T}[\cdot] \\ & | & \mathbb{T}[\cdot]\ \oplus_{\mathrm{T}}\ t \\ & | & \text{TVal}\ v \oplus_{\mathrm{T}} \mathbb{T}[\cdot] \\ & | & \text{TIf}\ \mathbb{T}[\cdot]\ t\ t' \\ & | & \text{TOrElse}\ \mathbb{T}[\cdot]\ t \end{array}$$

Evaluation contexts for processes

$$\begin{array}{lcl} \mathbb{P}[\cdot] & ::= & \text{PFork}\ \mathbb{P}[\cdot] \\ & | & \mathbb{P}[\cdot]\ \oplus_{\mathrm{P}}\ t \\ & | & \text{PVal}\ v\ \oplus_{\mathrm{P}} \mathbb{P}[\cdot] \\ & | & \text{PAt}\ (\text{Just}\ (i,j))\ \mathbb{P}[\cdot] \end{array}$$

Figure 3. Evaluation contexts for high-level language.

Transaction Semantics: We define transaction semantics by a reduction relation $\mapsto_{T_{ij}}$ on triples $\langle\Theta,\sigma,t\rangle$, where Θ is the current state of TM, σ is the history of TM execution and t is a transaction. Variables i and j denote the current transaction read stamp and id, respectively. First, we present the rule used to evaluate transaction contexts.

$$\frac{\langle\Theta,\ \sigma,\ t\rangle \mapsto_{T_{ij}} \langle\Theta',\ \sigma',\ t'\rangle}{\langle\Theta,\ \sigma,\ \mathbb{T}[t]\rangle \mapsto_{T_{ij}} \langle\Theta',\ \sigma',\ \mathbb{T}[t']\rangle}\ (TContext)$$

This rule simply allows stepping some subterm of current transaction expression $\mathbb{T}[t]$.

Next, we will consider how to evaluate a TRead construct. Notice that, we need two different rules for reading variables. This happens because, in our semantics, the function that reads a value from a variable can abort the current transaction, as it happens in TL2, if its write stamp is less than current transactions read stamp.

$$\frac{\Theta(v,i,j) = (val,\Theta') \quad \sigma' = \text{IRead}\ j\ v\ val\ : \sigma}{\langle\Theta,\ \sigma, \text{TRead}\ v\rangle \mapsto_{T_{ij}} \langle\Theta',\ \sigma',\ \text{TVal}\ val\rangle}\ (TReadOk)$$

$$\frac{\Theta(v,i,j) = (\text{TAbort},\Theta') \quad \sigma' = \text{IAbort}\ j\ : \sigma}{\langle\Theta,\ \sigma, \text{TRead}\ v\rangle \mapsto_{T_{ij}} \langle\Theta,\ \sigma',\ \text{TAbort}\rangle}\ (TReadFail)$$

Writing a value is done by next rules: rule $(TWriteVal)$ writes a completely reduced value and rule $(TWriteFail)$ just does propagate failure for signaling that current transaction has failed or aborted throught an explicit TRetry.

$$\frac{\begin{array}{c}\Theta' = \langle\Theta_h, \Theta_r, \Theta_w[j, x \mapsto val], \Theta_T\rangle \\ \sigma' = \text{IWrite}\ j\ v\ val\ : \sigma\end{array}}{\langle\Theta,\ \sigma, \text{TWrite}\ v\ (\text{TVal}\ val)\rangle \mapsto_{T_{ij}} \langle\Theta',\ \sigma',\ \text{TVal}\ val\rangle}\ (TWriteVal)$$

$$\frac{}{\langle\Theta,\ \sigma, \text{TWrite}\ v\ \text{TFail}\rangle \mapsto_{T_{ij}} \langle\Theta,\ \sigma,\ \text{TFail}\rangle}\ (TWriteFail)$$

Since we replace monadic bind by addition, we need to force a sequential order of evaluation and some additional rules to ensure the correct propagation of failure.

$$\frac{val = val_1 + val_2}{\langle \Theta,\ \sigma,\ (\mathsf{TVal}\, val_1) \oplus_{\mathsf{T}} (\mathsf{TVal}\, val_2)\rangle \mapsto_{T_{ij}} \langle \Theta,\ \sigma,\ \mathsf{TVal}\, val\rangle}\ (\mathit{TAddVal})$$

$$\frac{}{\langle \Theta,\ \sigma,\ \mathsf{TFail} \oplus_{\mathsf{T}} t\rangle \mapsto_{T_{ij}} \langle \Theta',\ \sigma', \mathsf{TFail}\rangle}\ (\mathit{TAddL})$$

$$\frac{}{\langle \Theta,\ \sigma,\ (\mathsf{TVal}\, val) \oplus_{\mathsf{T}} \mathsf{TFail}\rangle \mapsto_{T_{ij}} \langle \Theta',\ \sigma',\ \mathsf{TFail}\rangle}\ (\mathit{TAddR})$$

We can evaluate a TIf to its first branch if its condition is equal to zero or to its second otherwise.

$$\frac{}{\langle \Theta, \sigma, \mathsf{TIf}\ (\mathsf{TVal}\ 0)\ t\ t'\rangle \mapsto_{T_{ij}} \langle \Theta, \sigma,\ t\rangle}\ (\mathit{TIfZero})$$

$$\frac{v \neq 0}{\langle \Theta, \sigma, \mathsf{TIf}\ (\mathsf{TVal}\ \mathrm{v})\ t\ t'\rangle \mapsto_{T_{ij}} \langle \Theta, \sigma,\ t'\rangle}\ (\mathit{TIfNonZero})$$

We also propagate failures produced on TIf condition evaluation.

$$\frac{}{\langle \Theta,\ \sigma,\ \mathsf{TIf}\, \mathsf{TFail}\, t\, t'\rangle \mapsto_{T_{ij}} \langle \Theta,\ \sigma,\ \mathsf{TFail}\rangle}\ (\mathit{TIfFail})$$

Evaluating a TOrElse construct returns a value, if whenever its left transaction reduces to TVal v. Right transaction is executed only when the left one reduces to TRetry. Finally, if a transaction aborts such aborting signal is propagated.

$$\frac{}{\langle \Theta,\ \sigma,\ \mathsf{TOrElse}\ (\mathsf{TVal}\ v)\ t'\rangle \mapsto_{T_{ij}} \langle \Theta',\ \sigma',\ \mathsf{TVal}\ v\rangle}\ (\mathit{TOrElseVal})$$

$$\frac{}{\langle \Theta,\ \sigma,\ \mathsf{TOrElse}\, \mathsf{TRetry}\ t'\rangle \mapsto_{T_{ij}} \langle \Theta,\ \sigma,\ t'\rangle}\ (\mathit{TOrElseR})$$

$$\frac{}{\langle \Theta,\ \sigma,\ \mathsf{TOrElse}\, \mathsf{TAbort}\, t'\rangle \mapsto_{T_{ij}} \langle \Theta,\ \sigma,\ \mathsf{TAbort}\rangle}\ (\mathit{TOrElseA})$$

Process Semantics: The semantics for processes, $\mapsto_P$, acts on 5-uples $\langle \Theta, \sigma, j, i, s\rangle$ consisting of a TM state Θ, a history of transaction execution σ, last transaction id used j, a global clock i and a process soup s.

We begin the presentation of process semantics with context rule, which allows steps of inner expressions.

$$\frac{\langle \Theta,\ \sigma,\ j,\ i,\ t\rangle \mapsto_P \langle \Theta',\ \sigma',\ j',\ i',\ t'\rangle}{\langle \Theta,\ \sigma,\ j,\ i, \mathbb{P}[p] : s\rangle \mapsto_P \langle \Theta,\ \sigma',\ j',\ i',\ \mathbb{P}[p'] : s\rangle}\ (\mathit{PContext})$$

Process soup are represented by a list of processes and its execution proceeds by pattern matching on the first element of such list. In order to allow non-determinism we introduce a rule for preemption.

$$\frac{\langle \Theta,\ \sigma,\ j,\ i,\ s\rangle \mapsto_P \langle \Theta',\ \sigma',\ j',\ i',\ s'\rangle}{\langle \Theta,\ \sigma,\ j,\ i,\ p : s\rangle \mapsto_P \langle \Theta',\ \sigma',\ j',\ i', p : s'\rangle}\ (\mathit{PPreempt})$$

Evaluating a PFork adds a process p to current soup returning 0.

$$\frac{s' = \mathsf{PVal}\, 0 : p : s}{\langle \Theta,\ \sigma,\ j,\ i,\ (\mathsf{PFork}\, p) : s\rangle \mapsto_P \langle \Theta,\ \ \sigma,\ j,\ i,\ s'\rangle}\ (\mathit{PFork})$$

As we did with transaction, process composition is done using addition.

$$\frac{v = v_1 + v_2}{\langle \Theta,\ \sigma,\ j,\ i,\ (\mathsf{PVal}\, v_1) \oplus_{\mathsf{P}} (\mathsf{PVal}\, v_2)\rangle \mapsto_P \langle \Theta,\ \sigma,\ j,\ i,\ \mathsf{PVal}\, v\rangle}\ (\mathit{Add1})$$

Finally, we now present the semantics for atomic blocks. Unlike previous works [17, 21], the semantics of atomic blocks do not follow the so-called stop-the-world-semantics. This design choice is justified by the fact that stop-the-world semantics naturally enjoys safety conditions like opacity and markability. Since our objective is to exploit failures in STM algorithms represented as small-step semantics of our simple transactional language, the proposed semantics reduces atomic blocks in a step-wise manner instead of using a multi-step approach.

The first rule for reducing a PAt block is presented below. It basically updates the TM state with empty read and write sets for the newly started transaction j, register it using IBegin j and reinsert process PAt (i, j) t at the end of process soup. Notice that, initially, every atomic block does not have its read stamp and transaction id. When a transaction t is started, we update its process to store its starting clock and transaction id.

$$\frac{\begin{array}{c}\Theta_1 = \langle \Theta_h, \Theta_r\ [j \mapsto \bullet], \Theta_w\ [j \mapsto \bullet], \Theta_T\ [j \mapsto t]\rangle \\ s' = s \mathbin{+\!\!\!+} [\mathsf{PAt}\ (i, j)\ t] \\ \sigma' = \mathsf{IBegin}\, j : \sigma\end{array}}{\langle \Theta,\ \sigma,\ j,\ i,\ \mathsf{PAt}\ ()\ t\ :\ s\rangle \mapsto_P \langle \Theta',\ \sigma',\ j + 1,\ i,\ s'\rangle}\ (\mathit{PAt1})$$

After initializing a transaction, its execution proceeds thanks to rules *PPrempt* and *PContext*. Whenever a transaction successfully reduces to a value and it had executed in a consistent view of memory, we can use next rule to commit its results to heap.

$$\frac{\begin{array}{cc} v = \mathsf{TVal}\, n & \mathit{consistent}(\Theta, i, j) \\ \sigma' = \mathsf{ICommit}\, j : \sigma & \Theta' = \langle \Theta'_h,\ \Theta_r\ |_j,\ \Theta_w\ |_j\rangle \\ \multicolumn{2}{c}{\Theta'_h = \Theta_h \uplus \Theta_w(j)}\end{array}}{\langle \Theta,\ \sigma,\ j,\ i,\ \mathsf{PAt}\, v\ :\ s\rangle \mapsto_P \langle \Theta',\ \sigma',\ j,\ i + 1,\ \mathsf{PVal}\, n\ :\ s\rangle}\ (\mathit{PAt2})$$

We first check consistency using $\mathit{consistent}(\Theta, i, j)$, register a successful commit in history σ using ICommit j and update TM state Θ by: 1) writing the write set contents of transaction j in the heap and 2) removing the read and write set of transaction j from TM state.

Whenever a transaction reduces to TRetry or TAbort (represented by TFail), it should be restarted. For this, we remove entries for the transaction j from transactions and read / write set mappings. Also, we reinsert a process with the original transaction in the process soup to allow the restarting of this transaction. This is specified by next rule.

$$\frac{\begin{array}{c}\Theta' = \langle \Theta_h, \Theta_r\ |_j, \Theta_w\ |_j, \Theta_T\ |_j\rangle \\ s' = s \mathbin{+\!\!\!+} \mathsf{PAt}\ ()\ t \quad\ t = \Theta_t(j)\end{array}}{\langle \Theta,\ \sigma,\ j,\ i,\ \mathsf{PAt}\, \mathsf{TFail}\ :\ s\rangle \mapsto_P \langle \Theta',\ \sigma,\ j + 1,\ i,\ s'\rangle}\ (\mathit{PAt3})$$

3.2.2 STM-Haskell Based Semantics

Essentially, the STM-Haskell based semantics is just a simplification of the previouly defined one in which we do not take into account the global clock to ensure consistency of transaction logs. This change simplifies both the semantics and their auxiliar definitions to read variables and check for consistency of TM state.

In Figure 4, we redefine the function for reading a value from a variable. Note that this is almost the definition of Figure 1 except that it does not use a global clock for version control of variables in read set.

$$\Theta(x, j) = \begin{cases} (v, \Theta) & \textit{if } \Theta_w(x, j) = (i, v) \\ (v, \Theta) & \textit{if } \Theta_w(x, j) = \bot, \Theta_r(x, j) \neq \bot, \\ & \quad \Theta_h(x) = (i, v) \\ (v, \Theta_r\,[j, x \mapsto (i, v)]) & \textit{if } \Theta_w(x) = \Theta_r(x) = \bot, \\ & \quad \Theta_h(x) = (i, v) \end{cases}$$

Figure 4. Reading a variable

Also, since we do not abort current transaction when reading a value from a inconsistent memory view, the rule $(TReadFail)$ isn't necessary in STM-Haskell based semantics. When writing values to variables, the only change needed is to increment variable's write stamp. Modified rule is presented below.

$$\frac{\begin{array}{c}\Theta' = \langle \Theta_h, \Theta_r, \Theta_w[j, x \mapsto (i', val)], \Theta_T \rangle \\ \sigma' = \mathsf{lWrite}\, j\, v\, val\ : \sigma\end{array}}{\langle \Theta,\ \sigma, \mathsf{TWrite}\, v\ (\mathsf{TVal}\, val) \rangle \mapsto_{T_{ij}} \langle \Theta',\ \sigma',\ \mathsf{TVal}\, val \rangle}\ (TWriteVal)$$

We also need to modify the consistency check. In the original STM-Haskell paper [17], consistency of TM state is tested before a commit in order to validade if a transaction has accessed a valid memory view. This validity test essentially checks pointer equalities for values in read set. Since in our model we have no notion of pointer, we use value equality for consistency check as in [21].

$$consistent(\Theta, j) = \forall x. \Theta_r(x, j) = \Theta_h(x)$$

Figure 5. Predicate for consistency of transaction logs

Semantics for transactions and processes are essentially the same presented in previous section. Rules will differ only by: 1) Information about TL2 global clock isn't present and 2) it uses the modified consistency check and reading values from the TM state function presented in this section. For space reasons, we do not present this slightly modified set of semantic rules.

4 A Transactional Virtual Machine

4.1 Instruction Set

Our instruction set extends the one proposed by [21] with constructions to support conditionals, orElse and retry constructs. Conditionals are compiled using jump instructions and orElse and retry use instructions that "store" on VM stack code for exception handling, following the approach used by Hutton et.al. [23]. The syntax of our instruction set is defined by the following data type.

```
data Instr
  = Push Val | Add | Read Var | Write Var
  | Begin | Commit | Fork Code
  -- new instructions
  | Jump Offset | JumpFalse Offset | Retry
  | Mark Code | Unmark
```

Operation Push places a value on the machine stack and Add pops two values from stack and pushes its sum. Instruction Read pushes the value of a variable onto the stack and Write updates the variable with topmost value in the VM stack. Instructions Begin and Commit mark the start and finish of transactions and Fork spawns a new process. Instruction Retry is used to compile a call to TRetry and Mark places Retry handling code in VM stack while Unmark is the instruction responsible to remove a handler code from stack. Both Mark and Unmark are necessary to compile TOrElse construct.

Type Offset is just a synonym for Int and denotes how much should we move to execute the next instruction. Data type Code is just a zipper [22] for a list of Instr.

```
data Code = Code {
  executed :: [Instr]
  , nexts :: [Instr]
  }
```

In this enconding of list zippers, the current value on cursor is the head of nexts field. In VM semantics we represent values of data type code as a pair $\langle exs, nxs \rangle$ of Instr lists. This simplifies the presentation of how the semantics deal with control structures.

4.2 Compiler

In this section we present functions compileProc and compileTran that translate processes and transactions to VM instructions.

```
compileProc :: Proc → [Instr]
compileProc (PVal v)
  = [Push v]
compileProc (x ⊕P y)
  = compileProc x ++
    compileProc y ++ [Add]
compileProc (PAt t)
  = Begin : (compileTran t) ++ [Commit]
compileProc (PFork p)
  = [Fork (compileProc p)]
```

Compilation of transactions proceeds as follows: To compile a TRetry we just build a list containing a Retry instruction and conditionals are compiled using jumps and their offset is calculated from the result of compiling then and else branches. To compile a transaction t 'TOrElse' t1 we generate a Mark instruction containing the code built from t1, that will be put on VM stack and executed only if code for t reduces to TRetry, followed by compiled code for t and a trailing Unmark instruction, to remove handler code from stack, if t executes without reducing to a Retry.

```
compileTran :: Tran → [Instr]
compileTran (TVal i)
  = [Push i]
compileTran (x ⊕T y)
  = compileTran x ++ compileTran y ++ [Add]
compileTran (TRead v)
  = [Read v]
compileTran (TWrite v t)
  = compileTran t ++ [Write v]
compileTran TRetry
  = [Retry]
```

```
compileTran (t `TOrElse` t′)
  = [Mark (Code [ ] (compileTran t′))] ++
    compileTran t ++
    [Unmark]
compileTran (TIf t t′ t′′)
  = compileTran t ++
    [JumpFalse off1] ++
    c1 ++ [Jump off2] ++ c2
    where
      c1 = compileTran t′
      off1 = length c1 + 1
      c2 = compileTran t′′
      off2 = length c2
```

4.3 Virtual Machine Semantics

The VM semantics is specified as a transition relation $\mapsto_{VM}$ between 4-uples $\langle \Theta_h, \sigma, j, s \rangle$, comprising a heap Θ_h, a history σ, next transaction id j and a thread soup s. In turn, threads are represented by 5-uples of the form $\langle c,\ \rho,\ f,\ \Theta_r,\ \Theta_w \rangle$, where c is the code to be executed, ρ is the thread local stack, f is the code to be re-executed in case of failure and Θ_r and Θ_w are read and write-sets, respectively. Type StkCell represents an entry on a thread stack which can be a value or code to be executed whenever a Retry instruction is found during VM execution. We allow ourselves a bit of informality and represent values of type StkCell without explicitly mark them with constructors SVal and Han. We use variables n and m to denote SVal values and c, c' to denote Han. The main motivation is to avoid clutter in presenting VM semantics.

```
data StkCell
  = SVal Val
  | Han Code
type Stack = [StkCell]
```

The first rule of the VM semantics is to allow preemption, giving rise to non-deterministic execution.

$$\frac{\langle \Theta_h,\ \sigma, j\ s \rangle \mapsto_{VM} \langle \Theta'_h,\ \sigma', j', s' \rangle}{\langle \Theta_h,\ \sigma,\ j,\ t : s \rangle \mapsto_{VM} \langle \Theta'_h, \sigma',\ j',\ t : s' \rangle}\ (\mathit{Preempt})$$

The semantics of Fork adds a new thread to soup with empty read and write-sets.

$$\frac{\begin{array}{cc} C = \langle exs, \text{Fork}\ c' : c \rangle & C' = \langle \text{Fork}\ c' : exs, c \rangle \\ t = \langle C', 0 : \rho,\ f,\ \Theta_r,\ \Theta_w \rangle & t' = \langle \langle [], c' \rangle, [], [], \bullet, \bullet \rangle \end{array}}{\langle \Theta_h,\ \sigma, j,\ \langle C,\ \rho,\ f,\ \Theta_r,\ \Theta_w \rangle : s \rangle \mapsto_{VM} \langle \Theta_h,\ \sigma, j,\ t : t' : s \rangle}\ (\mathit{Fork})$$

Instructions Push and Add modify the stack by putting a new value on top of the stack and poping two values and pushing their sum on the thread local stack.

$$\frac{\begin{array}{c} C = \langle exs, \text{Push}\ n : c \rangle \\ t = \langle \langle \text{Push}\ n : exs, c \rangle,\ n : \rho,\ f,\ \Theta_r,\ \Theta_w \rangle \end{array}}{\langle \Theta_h,\ \sigma, j,\ \langle C,\ \rho,\ f,\ \Theta_r,\ \Theta_w \rangle : s \rangle \mapsto_{VM} \langle \Theta_h,\ \sigma, j,\ t : s \rangle}\ (\mathit{Push})$$

$$\frac{\begin{array}{c} C = \langle exs, \text{Add} : c \rangle \\ t = \langle C,\ n : m : \rho,\ f,\ \Theta_r,\ \Theta_w \rangle \\ t' = \langle c,\ (n+m) : \rho,\ f,\ \Theta_r,\ \Theta_w \rangle \end{array}}{\langle \Theta_h,\ \sigma, j,\ t : s \rangle \mapsto_{VM} \langle \Theta_h,\ \sigma, j,\ t' : s \rangle}\ (\mathit{Add})$$

Begin starts a new transaction execution by cleaning the read and write sets, setting code that should be executed when a failure happens, register in history σ the start of transaction and increments the transaction.

$$\frac{\begin{array}{c} t = \langle \langle exs, \text{Begin} : c \rangle, \rho, f, \Theta_r, \Theta_w \rangle \\ t' = \langle \langle \text{Begin} : exs, c \rangle, [], \text{Begin} : c, \bullet, \bullet \rangle \end{array}}{\langle \Theta_h, \sigma, j, t : s \rangle \mapsto_{VM} \langle \Theta_h, \sigma', j+1, t' : s \rangle}\ (\mathit{Begin})$$

Read pushes the value of variable v on top of the stack. First it consults the write log. If the variable has not been written to, the read log is then consulted. Otherwise, if the variable has also not been read, its value is looked up from the heap and the read log updated accordingly. In all situations described, a read for the variable is recorded.

$$\frac{\begin{array}{c} \sigma' = \text{IRead}\ (j-1)\ v\ n\ : \sigma \\ t = \langle \langle exs, \text{Read}\ v : c \rangle, \rho, f, \Theta_r, \Theta_w \rangle \\ t' = \langle \langle \text{Read}\ v : exs, c \rangle, n : \rho, f, \Theta'_r, \Theta_w \rangle \end{array}}{\langle \Theta_h, \sigma, j, t : s \rangle \mapsto_{VM} \langle \Theta_h, \sigma', j, t' : s \rangle}\ (\mathit{Read})$$

$$\langle n, \Theta'_r \rangle = \begin{cases} \langle \Theta_w(v), \Theta_r \rangle & \text{if } v \in dom(\Theta_w) \\ \langle \Theta_r(v), \Theta_r \rangle & \text{if } v \in dom(\Theta_r) \\ \langle \Theta_h(v), \Theta_r\ [v \mapsto \Theta_h(v)] \rangle & \text{otherwise.} \end{cases}$$

Operation Commit is responsible for finishing the execution of a transaction. When processing a commit, the VM updates the heap using the write-set if each variable in current thread read-set is equal to its value in the heap. Otherwise, current transaction should restart.

$$\frac{\begin{array}{c} \sigma' = e : \sigma \\ t' = \langle \langle exs, c' \rangle, \rho', f, \Theta_r, \Theta_w \rangle \\ t = \langle \langle exs, \text{Commit} : c \rangle, n : \rho, f, \Theta_r, \Theta_w \rangle \end{array}}{\langle \Theta_h, \sigma, j, t : s \rangle \mapsto_{VM} \langle \Theta'_h, \sigma', j, t' : s \rangle}$$

$$\langle \Theta'_h, c', \rho', e \rangle = \begin{cases} \langle \Theta_h \uplus \Theta_w, c, n : \rho, \text{ICommit}\ (j-1) \rangle & \text{if } \Theta_r \subseteq \Theta_h \\ \langle \Theta_h, f, \rho, \text{IAbort}\ (j-1) \rangle \end{cases}$$

Now, we consider instructions used to support conditionals, orElse and retry. Instructions for compiling conditionals, Jump and JumpFalse, are implemented using function jump which moves the position of next instruction to be executed accordingly.

```
jump :: Offset → Code → Code
jump off (Code os is)
  | off < 0 = Code os2 (os1 ++ is)
  | otherwise = Code (is1 ++ os) is2
  where
    off′ = abs off
    (is1, is2) = splitAt off′ is
    (os1, os2) = splitAt off′ os
```

In jump definition, whenever an offset off is negative we remove off elements of the Code executed field and add them to nexts field. When offset is positive we remove from nexts and insert them on executed.

Semantics of Jump and JumpFalse only differ in that the latter jumps only if the topmost element of current thread local stack is a non-zero integer.

$$\frac{\begin{array}{c} t = \langle\, \langle exs, \mathsf{Jump}\; n : c\rangle, \rho, f, \Theta_r, \Theta_w\rangle \\ t' = \langle \mathsf{jump}\; \mathsf{n}\, \langle exs, \mathsf{Jump}\; n : c\rangle, \rho, f, \Theta_r, \Theta_w\rangle \end{array}}{\langle \Theta_h,\ \sigma,\ j, t : s\rangle \mapsto_{VM} \langle \Theta_h,\ \sigma,\ j, t' : s\rangle}\ (Jump)$$

$$\frac{\begin{array}{c} t = \langle\, \langle exs, \mathsf{JumpFalse}\; n : c\rangle, m : \rho, f, \Theta_r, \Theta_w\rangle \\ t' = \langle C, \rho, f, \Theta_r, \Theta_w\rangle \\ C = \begin{cases} \langle \mathsf{JumpFalse}\; n : exs, c\rangle & \text{if } m = 0 \\ \mathsf{jump}\; \mathsf{n}\, \langle exs, \mathsf{Jump}\; n : c\rangle & \text{otherwise} \end{cases} \end{array}}{\langle \Theta_h,\ \sigma,\ j, t : s\rangle \mapsto_{VM} \langle \Theta_h,\ \sigma,\ j, t' : s\rangle}\ (JumpFalse)$$

Instruction Mark pushes handler code and Unmark removes a handler from current thread local stack.

$$\frac{\begin{array}{c} t = \langle\, \langle exs, \mathsf{Mark}\; c' : c\rangle, \rho, f, \Theta_r, \Theta_w\rangle \\ t' = \langle\, \langle \mathsf{Mark}\; c' : exs, c\rangle, c' : \rho, f, \Theta_r, \Theta_w\rangle \end{array}}{\langle \Theta_h,\ \sigma,\ j, t : s\rangle \mapsto_{VM} \langle \Theta_h,\ \sigma,\ j, t' : s\rangle}\ (Mark)$$

$$\frac{\begin{array}{c} t = \langle\, \langle exs, \mathsf{Unmark} : c\rangle, x : c' : \rho, f, \Theta_r, \Theta_w\rangle \\ t' = \langle\, \langle \mathsf{Unmark} : exs, c\rangle, x : \rho, f, \Theta_r, \Theta_w\rangle \end{array}}{\langle \Theta_h,\ \sigma,\ j, t : s\rangle \mapsto_{VM} \langle \Theta_h,\ \sigma,\ j, t' : s\rangle}\ (Unmark)$$

Intuitivelly, when the VM executes a Retry it just looks for a handler code in its stack and executes such code. To implement such behavior we use functions handler, which searches for a handler on thread's stack and skip, which removes remaining transaction instructions from threads code, by stopping when it finds a Unmark.

```
handler :: Stack → Code → (Code, Stack)
handler [ ] c
  = (c, [ ])
handler (SVal _ : s) c
  = handler s c
handler (Han c′ : s) c
  = (c {nexts = (nexts c′) ++ (nexts c)}, s)
skip :: Code → Code
skip c
  = Code (executed c) (skip (nexts c))
    where
      skip [ ] = [ ]
      skip (Unmark : nx) = nx
      skip (Mark _ : nx) = skip (skip nx)
      skip (_ : nx) = skip nx
```

Using these functions, the semantics of Retry is immediate. First, we use skip to remove from threads code the remaining instructions from current transaction and use handler to add handler code present in stack with the rest of current thread code.

$$\frac{\begin{array}{c} t' = \langle\, c', \rho', f, \Theta_r, \Theta_w\rangle \\ \langle c', \rho'\rangle = \mathsf{handler}\; \rho\, (\mathsf{skip}\, \langle exs, c\rangle) \\ t = \langle\, \langle exs, \mathsf{Retry} : c\rangle, x : c' : \rho, f, \Theta_r, \Theta_w\rangle \end{array}}{\langle \Theta_h,\ \sigma,\ j, t : s\rangle \mapsto_{VM} \langle \Theta_h,\ \sigma,\ j, t' : s\rangle}\ (Retry)$$

5 Safety Properties

Several safety conditions for TM were proposed in the literature, such as opacity [12], VWC [24], TMS1 and TMS2 [9] and markability [31]. All these conditions define indistinguishably criteria and the set of correct histories generated by the execution of TM. In this section, we present definitions of opacity and markability and describe Haskell implementations of these properties.

Before we give both definition and implementations of these criteria, we need to define some concepts. We say that a transaction is *live* in a history H if it has no commit or abort registered in H, otherwise it is *finished*. A history is said to be *legal* if all values read after a write in transactional variable are equal to last value written.

5.1 Opacity

Intuitively, if a TM algorithm has the opacity property it means that all histories produced by it are legal and preserves the real time ordering of execution, i.e. if a transaction T_i commits and updates a variable x before T_j starts then T_j cannot observe that old state of x. Guerraoui et.al. define formally opacity and provide a graph-based characterization of such property in a way that an algorithm is opaque only if the graph built from algorithm histories structure is acyclic [12]. In this work, we use a more direct encoding of opacity by representing it as a predicate over histories. We implement such predicate as a Haskell function following the textual definition present in [13].

We say that a TM algorithm is opaque if all prefixes of histories generated by it are final state opaque. Our Haskell definition of opacity is as follows:

```
opacity :: History → Bool
opacity = all finalStateOpacity ∘ inits
```

Function all checks if all elements of input list (second parameter) satisfy a predicate (first paremeter) and inits returns a list with all prefixes of a given list.

Our next step is to define when a history is final state opaque. We say that a finite history is final state opaque if exists some completion of it that preserves real time order and all of its transactions are legal. In Haskell code:

```
finalStateOpacity :: History → Bool
finalStateOpacity
  = some prop ∘ completions
    where
      prop tr = preservesOrder tr ∧ legal tr
      some p xs = (null xs) ∨ (any p xs)
```

Function completions produces a list of all completions of a given history. The completion of a history H is another history S, such that:

- All live and non-commit pending transactions of H are aborted in S; and
- All commit pending transactions of H are aborted or committed in S.

Since in our model we do not consider commit-pending transactions, completion of a history consists of aborting all live transactions. In order to abort all live transactions we have to split a history in sub-histories that group operations by transactions. This is done by function splits, which build a map between transaction ids and history items and return a list of histories, one for each transaction.

```
splits :: History → [History]
splits
  = Map.elems ∘ foldr step Map.empty
```

```
    where
      step i ac
          = maybe (Map.insert (stampOf i) [i] ac)
            (λis → Map.insert (stampOf i)
                (i : is)
                ac)
            (Map.lookup (stampOf i) ac)
```

Using splits, the definition of completions is immediate: we just abort each non-committed transaction and remove them together with failed ones. Checking if a sub-history for a transaction is comitted or not is a simple check if the last item of sub-history is equal to ICommit or not.

```
completions :: History → [History]
completions
    = foldr abortLives [ ] ∘ splits
      where
        abortLives tr ac
            | finished tr = tr : ac
            | otherwise = ac
completed :: History → Bool
completed
    = finished ∘ last
      where
        finished (ICommit _) = True
        finished (IAbort _) = True
        finished _           = False
```

To finish the implementation of finalStateOpacity, we need to present definitions of preservesOrder and legal. The function that verifies if a history preserves *real time ordering* is preservesOrder. Let t_k and t_m be transactions of some history H. We say that $t_k \prec_H t_m$, if whenever t_k is completed and the last event of t_k precedes the first event of t_m in H. A history H' preserves the real time ordering of H if for all transactions t_k and t_m, if $t_k \prec_H t_m$ then $t_k \prec_{H'} t_m$. Intuitively, function preservesOrder checks if transaction ids are ordered according to its position in history.

```
preservesOrder :: History → Bool
preservesOrder tr
    = and [i ⩽ i′ | (p, i) ← tr′,
      (p′, i′) ← tr′,
      p ⩽ p′]
      where
        tr′ :: [(Int, Stamp)]
        tr′ = zipWith step [0 . .] tr
        step p i = (p, (stampOf i))
```

In order to check if all events of a transaction are legal we build a map between variables and events of read and writing to them using function sequentialSpecs which, in turn, uses function readOrWrite that returns a variable plus the event itself or Nothing, if it was not a read or write event.

```
readOrWrite :: Event → Maybe (Var, Event)
readOrWrite i@(IRead _ v _)
    = Just (v, i)
readOrWrite i@(IWrite _ v _)
    = Just (v, i)
readOrWrite _
    = Nothing
sequentialSpecs :: History → [History]
sequentialSpecs
    = Map.elems ∘ step1 ∘ mapMaybe readOrWrite
      where
        step1 = foldr step Map.empty
        step (v, i) ac
            = maybe (Map.insert v [i] ac)
              (λis → Map.insert v (i : is) ac)
              (Map.lookup v ac)
```

Finally, function legal checks if all values read for a variable are equal to last value written, by folding over the list of events built for each variable by function sequentialSpecs.

```
legal :: History → Bool
legal
    = all isLegal ∘ sequentialSpecs
      where
        isLegal = fst ∘ foldr step (True, Map.empty)
        step (IRead _ v val) (c, m)
            = maybe (val ≡ (Val 0), m)
              ((, m) ∘ (c ∧) ∘ (≡ val))
              (Map.lookup v m)
        step (IWrite _ v val) (c, m)
            = (c, Map.insert v val m)
        step _ ac = ac
```

Opacity can be characterized by building a graph over the set of generated histories by a TM algorithm. Such proof for the TL2 algorithm can be found in [13].

5.2 Markability

Another criteria for TM safety is markability [31], which decomposes opacity in a conjunction of three invariants, namely: 1) write-observation, which requires that each read operation returns the most current value; 2) read-preservation requires that the read location is not overwritten in the interval that the location is read and the transaction takes effect and 3) real-time preservation property, already present in opacity. Also, Lesani et. al. [31] proves that markability is equivalent to opacity.

A history is said to be *final state markable* if there is an extension of it that is write-observant, read-preserving and preserves real-time. An extension of a history is obtained by committing or aborting its commit-pending transactions and aborting the other live transactions. Our Haskell implementation of markability is just the conjunction of these previous properties and uses function completions that generates a list of events grouped by transaction.

```
markability :: History → Bool
markability h
    = and [writeObservation h′
      , readPreservation h′
      , preservesOrder h′]
        where
          h′ = concat $ completions h
```

Function writeObservation checks if a history is write-observant. It builds a mapping between variables and last values written and traverses the history verifying if subsequent event read the most recent value written.

```
writeObservation :: History → Bool
writeObservation
  = Map.foldr ((∧) ∘ snd) True m
    where
      m = foldr step Map.empty h
      step (IRead _ v val) m
          = case Map.lookup v m of
              Nothing → Map.insert v ((Val 0), True) m
              Just (val′, ac) → Map.insert v (val, ac ∧ val ≡ val′) m
      step (IWrite _ v val) m
          = Map.insert v (val, True) m
      step _ ac = ac
```

The last piece for our implementation of a markability test is function readPreservation, which builds a mapping between variables and values in order to ensure a value read by a transaction isn't overwritten before it commits.

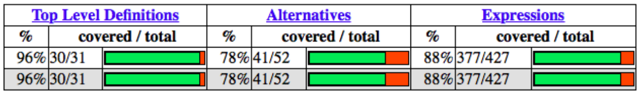
```
readPreservation :: History → Bool
readPreservation h
  = all (fst ∘ foldr step (True, Map.empty)) h′
    where
      step (IRead _ v val) (ac, m)
          = case Map.lookup v m of
              Nothing → (ac, Map.insert v (Val 0) m)
              Just val′ → (ac ∧ val ≡ val′, Map.insert v val′ m)
      step (IWrite _ v val) (ac, m)
          = case Map.lookup v m of
              Nothing → (ac, Map.insert v val m)
              Just val′ → (ac, m)
      step _ ac = ac
      h′ = [l | l ← splits h
         , isCommit (last l)]
      isCommit (ICommit _) = True
      isCommit _ = False
```

6 Validation of Semantic Properties

After the presentation of language semantics and the implementation of STM safety properties in terms of execution histories, how can we be sure that the defined semantics enjoys and the compiler preserves these properties? We follow the lead of [21] and use QuickCheck [3] to generate random high-level programs and check them against the following properties:

1. Does histories produced by TL2-based and STM-Haskell-based semantics have the opacity property?
2. Does histories by TL2-based and STM-Haskell-based semantics have the markability property?
3. Does the compiler preserves these safety properties?
4. Are the presented semantics equivalent to traditional stop-the-world-semantics for STM?

Each of these properties have been implemented as Haskell functions and tested using QuickCheck for randomly test cases. Having running many thousands of tests, we gain a high degree of confidence in the safety of our semantics, but it is important to measure how much of code base is covered by the test suite. Such statistics are provided by Haskell Program Coverage tool [10]. Results of code coverage are presented in the next figure.



Top Level Definitions			Alternatives			Expressions		
%	covered / total		%	covered / total		%	covered / total	
96%	30/31	🟩🟥	78%	41/52	🟩🟥	88%	377/427	🟩🟥
96%	30/31	🟩🟥	78%	41/52	🟩🟥	88%	377/427	🟩🟥

Figure 6. Test Coverage Results

While not having 100% of code coverage, our test suite provides a strong evidence that proposed semantics enjoys safety properties by exercising on randomly generated programs of increasing size. By analysing test coverage results, we can observe that code not reached by test cases consists of stuck states on program semantics.

For generating random programs we use basic generators provided by QuickCheck library and build Arbitrary instances for Tran and Proc types. Below, we present a snippet of instance for Proc. Code for Tran follows the same structure.

```
instance Arbitrary Proc where
  arbitrary
     = sized genProc
genProc :: Int → Gen Proc
genProc n
  | n ⩽ 0 = PVal ⟨$⟩ arbitrary
  | otherwise
        = frequency
          [
             (n + 1, PVal ⟨$⟩ arbitrary)
          , (n2, PFork ⟨$⟩ genProc (n − 1))
          , (n2, PAt Nothing ⟨$⟩ arbitrary)
          , (n, ( ⊕P ) ⟨$⟩ genProc n2 ⟨*⟩ genProc n2)
          ]
    where
      n2 = div n 2
```

The sized function allows for generating values with a size limit and frequency creates a generator that chooses each alternative with a probability proportional to the accompanying weight.

The TL2-based semantics passed in all tests for safety properties, as expected, since it is well-known that TL2 provides opacity. But, the semantics based on STM-Haskell does not enjoy such safety properties since it allows the reading from an inconsistent view of memory. Next example shows how such invalid memory access can happen.

Example 2. Consider the following program, where x is some variable:

```
t1 :: Tran
t1 = TRead x ⊕T TRead x ⊕T TRead x

t2 :: Tran
t2 = TWrite x v

p :: Proc
p = Fork (PAt Nothing t1) ⊕P Fork (PAt Nothing t2)
```

One of the possible executions of p using STM-Haskell semantics would result in the following history:

```
[IBegin 1, IBegin 2, IRead 1 x 0
, IWrite 2 x 10, IRead 1 x 0, ICommit 2
, IRead 1 x 0, ...]
```

which violates opacity and markability because it does allow transaction t1 to read from an inconsistent memory view. On TL2 semantics safety is preserved because when transaction t1 tries to execute third read it would be aborted.

7 Related Work

Semantics for STM: Semantics for STM have been received a considerable attention recently [1, 17, 32, 33]. Harris et al. [17] defines a stop-the-world operational semantics for STM Haskell. Essentially, Harris uses a multi-step execution model for transaction execution that does not allows the investigation of safety property neither how interleaving of transactions happens. Such approach for STM semantics does not allows the investigation of safety properties in terms of execution histories, since no interleaving between transactions happen.

Abadi et. al. [1] developed the so-called calculus of automatic mutual exclusion (AME) and shows how to model the behavior of atomic blocks. Using AME they model STM systems that use in-place update, optimistic concurrency, lazy-conflict detection and roll-back and determine assumptions that ensure correctness criteria. As [1], our work defines different semantics for the same language with the intent to verify STM algorithms, but they use manual proofs to assert that their semantics enjoy criteria of interest and our work advocates the use of automated testing tools to early discover semantic design failures before starting proofs.

Moore et. al. [33] proposes a series of languages that model several behaviors of STM. Such models abstract implementation details and provide high-level definitions. Moore uses small-step operational semantics to explicitly model interleving execution of threads. Manual proofs of isolation properties are described as a techinical report [34].

Safety properties for STM: Safety criteria for STM was another line of research pursued recently [12, 31]. Opacity was defined by Guerraoui et. al. [12] and it is described as a condition on generated histories by a TM algorithm and provide a graph-based characterization of opacity. Such graph is built from histories and an algorithm is considered opaque if the corresponding graph is acyclic for every possible history. Lesani et. al. [31] describes an equivalent safety property called markability, which decomposes opacity in three invariants and prove that these invariants are equivalent to opacity.

Formal verification of STM: Formal verification of STM algorithms has been an active subject of recent research [4, 5, 11, 29, 30]. Lehsani et.al. [29] describes a PVS [36] formalization of a framework for verifying STM algorithms based on I/O automata. The main idea of Lehsani's framework is to represent both specifications and algorithms as automata and their equivalence is verified by proving a simulation relation between these automata. The use of model checker to verify TM algorithms was the subject of [4, 5]. Both works use the specification languages of model checkers [27] to describe STM implementations and check them against safety properties. We leave using proof assistants for verifying safety properties of our STM semantics for future work.

Testing algorithms for STM: Automated testing for a compiler of a STM high-level language to a virtual machine was the subject of [21]. He uses QuickCheck to generate random high-level STM programs and check that their virtual machine compiler preserves the semantics of source programs. Unlike our work that focus on verifying safety of algorithms expressed as small-step operational semantics, Hu et. al. concerns only with semantic preservation of compilation process and uses multi-steps to evaluate transactions in a stop-the-world semantics for their high-level language. While such semantics design choices are reasonable for verifying a semantic preservation theorem for a compiler, they do allow for checking safety properties. Harmanci et. al. [16] describes a tool for testing TM algorithms, called TM-unit. Using TM-unit domain specific language, users can specify TM workloads for both safety and performance testing of TM algorithms. Authors argue that their domain specific language is simple and expressive but no formal semantics of this language is provided. We believe that the use of domain specific languages is invaluable to provide concise and formal specifications of STM algorithm and we leave this line of research for further work.

8 Conclusion

In this work we presented safe semantics for a simplied high-level language with STM support and use property based testing to verify it. The lightweight approach provided by QuickCheck allow us to experiment with different semantic designs and implementations, and to quickly check any changes. During the development of this work, we have changed our basic definitions many times, both as a result of correcting errors, and streamlining the presentation. Ensuring that our changes were consistent was simply a matter of re-running test suite. Encoding safety properties as Haskell functions over STM histories provides a clean and concise implementation that helps not only to fix semantics but also to improve our understanding of STM algorithms.

As future work we intend to use Agda [35] to provide formally certified proofs that the presented semantics does enjoy safety properties and also investigate the usage of domain specific languages to ease the task of specifying algorithms as small-step operational semantics of a simple transactional language.

References

[1] Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. 2011. Semantics of Transactional Memory and Automatic Mutual Exclusion. *ACM Trans. Program. Lang. Syst.* 33, 1, Article 2 (Jan. 2011), 50 pages. DOI: http://dx.doi.org/10.1145/1889997.1889999

[2] Lee Baugh, Naveen Neelakantam, and Craig Zilles. 2008. Using Hardware Memory Protection to Build a High-Performance, Strongly-Atomic Hybrid Transactional Memory. *SIGARCH Comput. Archit. News* 36, 3 (June 2008), 115–126. DOI: http://dx.doi.org/10.1145/1394608.1382132

[3] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00).* ACM, New York, NY, USA, 268–279. DOI: http://dx.doi.org/10.1145/351240.351266

[4] Ariel Cohen, John W. O'Leary, Amir Pnueli, Mark R. Tuttle, and Lenore D. Zuck. 2007. Verifying Correctness of Transactional Memories. In *Proceedings of the 7th International Conference on Formal Methods in Computer-Aided Design (FMCAD).* 37–44.

[5] Ariel Cohen, Amir Pnueli, and Lenore D. Zuck. 2008. Mechanical Verification of Transactional Memories with Non-transactional Memory Accesses. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings (Lecture Notes in Computer Science),* Aarti Gupta and

Sharad Malik (Eds.), Vol. 5123. Springer, 121–134. DOI: http://dx.doi.org/10.1007/978-3-540-70545-1_13

[6] Luke Dalessandro, François Carouge, Sean White, Yossi Lev, Mark Moir, Michael L. Scott, and Michael F. Spear. 2011. Hybrid NOrec: A Case Study in the Effectiveness of Best Effort Hardware Transactional Memory. *SIGPLAN Not.* 46, 3 (March 2011), 39–52. DOI: http://dx.doi.org/10.1145/1961296.1950373

[7] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08).* Springer-Verlag, Berlin, Heidelberg, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

[8] Dave Dice, Ori Shalev, and Nir Shavit. 2006. Transactional Locking II. In *Proceedings of the 20th International Conference on Distributed Computing (DISC'06).* Springer-Verlag, Berlin, Heidelberg, 194–208. DOI: http://dx.doi.org/10.1007/11864219_14

[9] Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. 2009. Towards Formally Specifying and Verifying Transactional Memory. *Electron. Notes Theor. Comput. Sci.* 259 (Dec. 2009), 245–261. DOI: http://dx.doi.org/10.1016/j.entcs.2010.01.001

[10] Andy Gill and Colin Runciman. 2007. Haskell Program Coverage. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop (Haskell '07).* ACM, New York, NY, USA, 1–12. DOI: http://dx.doi.org/10.1145/1291201.1291203

[11] Rachid Guerraoui, Thomas A. Henzinger, Barbara Jobstmann, and Vasu Singh. 2008. Model Checking Transactional Memories. *SIGPLAN Not.* 43, 6 (June 2008), 372–382. DOI: http://dx.doi.org/10.1145/1379022.1375626

[12] Rachid Guerraoui and Michal Kapalka. 2008. On the Correctness of Transactional Memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08).* ACM, New York, NY, USA, 175–184. DOI: http://dx.doi.org/10.1145/1345206.1345233

[13] Rachid Guerraoui and Michal Kapalka. 2010. *Principles of Transactional Memory.* Morgan-Claypool Publishers. http://dx.doi.org/10.2200/S00253ED1V01Y201009DCT004

[14] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. 2004. Transactional Memory Coherence and Consistency. *SIGARCH Comput. Archit. News* 32, 2 (March 2004), 102–. DOI: http://dx.doi.org/10.1145/1028176.1006711

[15] Ruud Haring, Martin Ohmacht, Thomas Fox, Michael Gschwind, David Satterfield, Krishnan Sugavanam, Paul Coteus, Philip Heidelberger, Matthias Blumrich, Robert Wisniewski, alan gara, George Chiu, Peter Boyle, Norman Chist, and Changhoan Kim. 2012. The IBM Blue Gene/Q Compute Chip. *IEEE Micro* 32, 2 (March 2012), 48–60. DOI: http://dx.doi.org/10.1109/MM.2011.108

[16] Derin Harmanci, Pascal Felber, Vincent Gramoli, and Christof Fetzer. 2009. TMUNIT: Testing Transactional Memories. In *TRANSACT '09: 4th Workshop on Transactional Computing.*

[17] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. 2005. Composable Memory Transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '05).* ACM, New York, NY, USA, 48–60. DOI: http://dx.doi.org/10.1145/1065944.1065952

[18] Maurice Herlihy, Victor Luchangco, and Mark Moir. 2006. A Flexible Framework for Implementing Software Transactional Memory. *SIGPLAN Not.* 41, 10 (Oct. 2006), 253–262. DOI: http://dx.doi.org/10.1145/1167515.1167495

[19] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. 2003. Software Transactional Memory for Dynamic-sized Data Structures. In *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing (PODC '03).* ACM, New York, NY, USA, 92–101. DOI: http://dx.doi.org/10.1145/872035.872048

[20] Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional Memory: Architectural Support for Lock-free Data Structures. *SIGARCH Comput. Archit. News* 21, 2 (May 1993), 289–300. DOI: http://dx.doi.org/10.1145/173682.165164

[21] Liyang Hu and Graham Hutton. 2009. Towards a Verified Implementation of Software Transactional Memory. In *Trends in Functional Programming volume 9*, Peter Achten, Pieter Koopman, and Marco Morazan (Eds.). Intellect. Selected papers from the Ninth Symposium on Trends in Functional Programming, Nijmegen, The Netherlands, May 2008.

[22] Gérard Huet. 1997. The Zipper. *J. Funct. Program.* 7, 5 (Sept. 1997), 549–554. DOI: http://dx.doi.org/10.1017/S0956796897002864

[23] Graham Hutton and Joel Wright. 2007. What is the Meaning of These Constant Interruptions? *Journal of Functional Programming* 17, 6 (Nov. 2007), 777–792.

[24] Damien Imbs, José Ramon de Mendivil, and Michel Raynal. 2009. Brief Announcement: Virtual World Consistency: A New Condition for STM Systems. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing (PODC '09).* ACM, New York, NY, USA, 280–281. DOI: http://dx.doi.org/10.1145/1582716.1582764

[25] Intel. 2012. *Intel Architecture Instruction Set Extensions Programming Reference.* Technical Report 319433-014. Intel. http://download-software.intel.com/sites/default/files/319433-014.pdf

[26] Eric Koskinen, Matthew Parkinson, and Maurice Herlihy. 2010. Coarse-grained Transactions. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10).* ACM, New York, NY, USA, 19–30. DOI: http://dx.doi.org/10.1145/1706299.1706304

[27] Leslie Lamport. 2002. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[28] Matthew Le, Ryan Yates, and Matthew Fluet. 2016. Revisiting Software Transactional Memory in Haskell. In *Proceedings of the 9th International Symposium on Haskell (Haskell 2016).* ACM, New York, NY, USA, 105–113. DOI: http://dx.doi.org/10.1145/2976002.2976020

[29] Mohsen Lesani, Victor Luchangco, and Mark Moir. 2012. A Framework for Formally Verifying Software Transactional Memory Algorithms. In *Proceedings of the 23rd International Conference on Concurrency Theory (CONCUR'12).* Springer-Verlag, Berlin, Heidelberg, 516–530. DOI: http://dx.doi.org/10.1007/978-3-642-32940-1_36

[30] Mohsen Lesani and Jens Palsberg. 2013. Proving Non-opacity. In *Proceedings of the 27th International Symposium on Distributed Computing - Volume 8205 (DISC 2013).* Springer-Verlag New York, Inc., New York, NY, USA, 106–120. DOI: http://dx.doi.org/10.1007/978-3-642-41527-2_8

[31] Mohsen Lesani and Jens Palsberg. 2014. Decomposing Opacity. In *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings (Lecture Notes in Computer Science)*, Fabian Kuhn (Ed.), Vol. 8784. Springer, 391–405. DOI: http://dx.doi.org/10.1007/978-3-662-45174-8_27

[32] Ben Liblit. 2006. An Operational Semantics for LogTM — Technical Report 1571. (2006). http://research.cs.wisc.edu/multifacet/papers/tr1571_logtm_semantics.pdf

[33] Katherine F. Moore and Dan Grossman. 2008. High-level Small-step Operational Semantics for Transactions. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08).* ACM, New York, NY, USA, 51–62. DOI: http://dx.doi.org/10.1145/1328438.1328448

[34] Katherine F. Moore and Dan Grossman. 2008. High-level Small-step Operational Semantics for Transactions (technical companion). (2008). http://homes.cs.washington.edu/~kfm/atomsfamily_proofs.pdf

[35] Ulf Norell. 2009. Dependently Typed Programming in Agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation (TLDI '09).* ACM, New York, NY, USA, 1–2. DOI: http://dx.doi.org/10.1145/1481861.1481862

[36] Sam Owre, John M. Rushby, and Natarajan Shankar. 1992. PVS: A Prototype Verification System. In *Proceedings of the 11th International Conference on Automated Deduction: Automated Deduction (CADE-11).* Springer-Verlag, London, UK, UK, 748–752. http://dl.acm.org/citation.cfm?id=648230.752639

[37] Victor Pankratius and Ali-Reza Adl-Tabatabai. 2011. A Study of Transactional Memory vs. Locks in Practice. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '11).* ACM, New York, NY, USA, 43–52. DOI: http://dx.doi.org/10.1145/1989493.1989500

[38] Nir Shavit and Dan Touitou. 1995. Software Transactional Memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '95).* ACM, New York, NY, USA, 204–213. DOI: http://dx.doi.org/10.1145/224964.224987