The Semantic of Twilight Transactions

Annette Bieniusa Peter Thiemann

March 24, 2011

1 Introduction

By avoiding observable inconsistencies, the semantics of transactional memory provides a comparatively simple model for concurrent programming. Instead of (implicitly) associating several memory locations with a lock and requiring that the lock needs to be obtained before accessing any of these memory locations and released thereafter, accesses are grouped together in a transaction that runs at a proclaimed level of isolation.

Prior work on the semantics of transactions [1, 6] focused primarily on weak atomicity, which is important for hybrid applications (for example, an application that includes legacy code using locking as well as new transactional code) because it helps to study the interaction of transactional and non-transactional memory accesses. However, these formalizations do not account for the phenomena that occur in an interleaved execution of transactions. For example, in state-of-the-art algorithms like TL2 [3], threads may get stuck even when a fair scheduling of threads is provided because they are repeatedly forced to abort by other transactions' successful commits.

To illustrate the mechanism underlying the aborts, this chapter pursues an approach that abstracts program execution by traces of memory accesses and transaction control operations. To this end, we define a monadic lambda calculus with threads and transactions, Λ_{STM}. Similar to schemes in research on isolation level for databases, each memory access is modeled by an effect on the global heap. This abstraction allows for an easy comparison of different TM algorithms. Their semantics are reflected in the effect traces which they generate during program execution under certain scheduling schemes. The traces are then used to show that the TM algorithms implement isolation levels like opacity or snapshot isolation.

This chapter is ordered as follows:

1. We present a formalization of a semantics for transactional memory that is suitable for proving properties of a TM implementation.

   A high-level semantics abstracts so many details that properties of the implementation become trivially evident [5]. A low-level semantics provides so many details that formal proof of its properties is no longer tractable. An example is the pseudocode for an implementation. Our semantics keeps the middle ground. It explicitly

models the non-deterministic interleaving of the operations in each thread including operations in aborting transactions. However, it does not model implementation details like the construction of memory snapshots or the implementation of locks.

2. We prove that our semantics for Λ_{STM} implements opacity [4], that is, all execution traces in our semantics are equivalent to serial execution traces, where the execution of critical regions, namely the transaction bodies, is non-interleaved.

3. We demonstrate that a small modification of the semantics (the TM algorithm, respectively) yields another notion of transactional isolation, namely snapshot isolation. We define a criterion for traces that suffice snapshot isolation and prove that the modified semantics Λ_{TWI} only produces such snapshot traces.

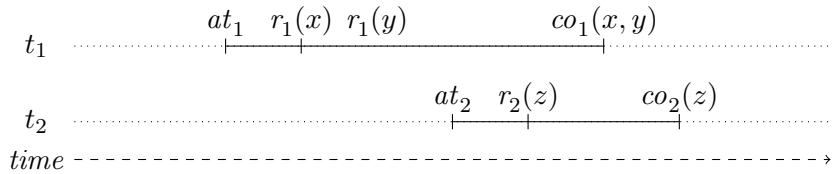
2 Execution traces

Let us begin with some examples of execution traces that provide insight into our approach. These traces abstract over atomic effects that denote the beginning of a transaction (at_i^t), read accesses to variable x within a transaction ($r_i^t(x)$), commits of a transaction which correspond to globally visible modifications to shared variables ($co_i^t(\bar{x})$), and abort effects for unsuccessful transactions (ab_i^t). In these effects, i is a (unique) transaction id and t the thread identifier.

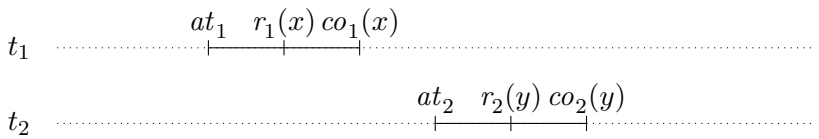
To simplify reasoning, we rely on an abstract notion of time. Effect are supposed happens atomically at a distinct, single point in time. Further, effects can be (totally) ordered according to the point in time when they occur.

2.1 Successful commits

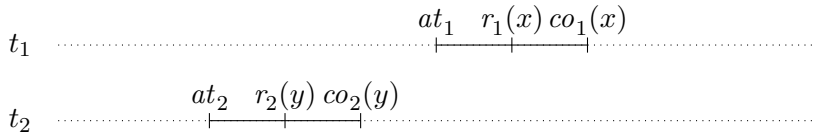
If all transactions in a program's run commit successfully, the execution trace can be turned into a serializable one by adapting the scheduler's decision. For example, consider this trace:



The scheduling interleaves two transactions t_1 and t_2 that read and write disjoint variables $x \neq y \neq z$. For this trace, there are two equivalent serial traces:



and

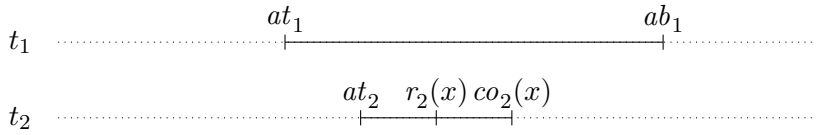


Both traces correspond to evaluations of the program to the same final heap and result. During the evaluation, each transaction conceptually operates on its own snapshot of the heap, taken at the beginning of the transaction. As both transactions were able to finish successfully, their read and write sets cannot have elements in common, and all their operations are independent. Hence both serial traces are equivalent to the original one.

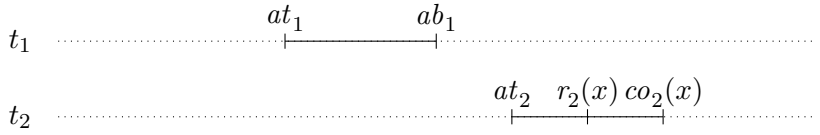
2.2 Read conflicts

A read conflict occurs if one transaction commits a write operation to a variable that another transaction is just about to read. In that case, the reading transaction must not proceed because its snapshot is no longer consistent with the current heap. Thus, the semantics forces the second transaction to abort.

An example of abort induced by a read conflict is depicted in the next trace.

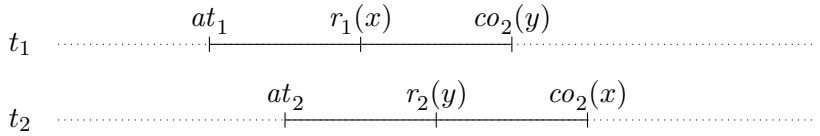


The trace gives an example for a transaction t_1 which had to abort because of a read conflict on variable x (it does not produce a read effect because the read operation is never permitted, as explained above). Nevertheless, there is an equivalent serial trace:



2.3 Snapshot isolation

Not all interleavings of threads can be decoupled into some equivalent serial trace. Take a look at the next example:



It is not serializable because a read operation is supposed to return the last value written to a variable. Hence, in a serial trace the latter operation would yield the value written by the first one.

Algorithms that allow traces like this implement a weaker isolation level called snapshot isolation. Semantically, a thread-local copy of the memory is made at the start of a transaction. The transaction operates this private copy during its execution. At commit,

Figure 1 Syntax of Λ_{STM}. Gray expressions arise only during evaluation.

$$
\begin{array}{lll}
x \in \mathrm{Var} & & \\
l \in \mathrm{Ref} & & \\
v \in \mathrm{Val} & ::= & l \mid \mathtt{tt} \mid \mathtt{ff} \mid () \mid \lambda x.e \mid \mathtt{return}\, e \\
e \in \mathrm{Exp} & ::= & v \mid x \mid e\ e \mid \mathtt{if}\ e\ e\ e \mid e \gg\!= e \\
 & & \mid \mathtt{spawn}\, e \mid \mathtt{atomic}\, e \mid (e, W_i, R_i, i, e, \mathcal{H}) \\
 & & \mid \mathtt{newref}\, e \mid \mathtt{readref}\, e \mid \mathtt{writeref}\, e\, e
\end{array}
$$

all changes are merged back into the global heap. The transaction is in conflict with another transactions only if both transactions try to update the same heap locations. A detailed discussion of snapshot isolation is done in Section 5.

3 Formalization

This section formalizes an STM with lazy update, where all write operations are delayed till the commit operation. The formalization is based on a monadic call-by-name lambda calculus with references, threads, and transactions.

3.1 Syntax

Figure 1 contains the syntax of Λ_{STM}. A value is either a reference, a boolean, the unit constant, or a function. Expressions comprise these values, variables, function application, conditional, monadic return and bind, spawning of threads, transactions, transactions in progress (an intermediate expression not arising in source programs), and the usual operations on references.

Figure 2 defines the type system for Λ_{STM}. The type language consists of the types of the simply typed lambda calculus with base types boolean and unit, a reference type $\mathtt{R}\,\tau$ for references pointing to values of type τ, function types, and a monadic type $\mu\,\tau$ for a monad returning values of type τ. There is a choice of two monads, `IO` for general monadic operations and `STM` for operations inside a transaction.

The typing judgment $\Sigma|\Gamma \vdash e : \tau$ contains two environments: Σ tracks the type of memory locations, and Γ tracks the type of variables. There is a second, heap typing judgment $\Sigma \vdash \mathcal{H}$ that relates the type of each memory location to the (closed) value stored in it. The typing rules are syntax-directed and mostly standard.

3.2 Operational Semantics

Figure 3 introduces some further definitions for the operational semantics. A program state $\mathcal{H}, \mathcal{P}$ is a pair consisting of a heap and a thread pool. A thread pool maps thread identifiers to expressions to be evaluated concurrently. The execution of a program is represented by a labeled transition relation between program states.

Figure 2 Typing rules of Λ_{STM}.

$$\text{Types} \qquad \begin{array}{rcl} \tau & ::= & \texttt{bool} \mid () \mid \texttt{R}\,\tau \mid \tau \to \tau \mid \mu\,\tau \\ \mu & ::= & \texttt{IO} \mid \texttt{STM} \end{array}$$

$$\frac{}{\Sigma|\Gamma \vdash \texttt{ff} : \texttt{bool}}\ \text{T-False} \qquad \frac{}{\Sigma|\Gamma \vdash \texttt{tt} : \texttt{bool}}\ \text{T-True} \qquad \frac{}{\Sigma|\Gamma \vdash () : ()}\ \text{T-Unit}$$

$$\frac{\Gamma(x) = \tau}{\Sigma|\Gamma \vdash x : \tau}\ \text{T-Var} \qquad \frac{\Sigma(l) = \tau}{\Sigma|\Gamma \vdash l : \texttt{R}\,\tau}\ \text{T-Ref}$$

$$\frac{\Sigma|\Gamma, x : \tau_1 \vdash e : \tau_2}{\Sigma|\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}\ \text{T-Func} \qquad \frac{\Sigma|\Gamma \vdash e_2 : \tau_1 \to \tau_2 \qquad \Sigma|\Gamma \vdash e_1 : \tau_1}{\Sigma|\Gamma \vdash e_2\ e_1 : \tau_2}\ \text{T-App}$$

$$\frac{\Sigma|\Gamma \vdash e_1 : \texttt{bool} \qquad \Sigma|\Gamma \vdash e_2 : \tau \qquad \Sigma|\Gamma \vdash e_3 : \tau}{\Sigma|\Gamma \vdash \texttt{if}\ e_1\ e_2\ e_3 : \tau}\ \text{T-If}$$

$$\frac{\Sigma|\Gamma \vdash e : \tau}{\Sigma|\Gamma \vdash \texttt{return}\ e : \mu\,\tau}\ \text{T-Return} \qquad \frac{\Sigma|\Gamma \vdash e_1 : \mu\,\tau \qquad \Sigma|\Gamma \vdash e_2 : \tau \to \mu\,\tau'}{\Sigma|\Gamma \vdash e_1 \gg\!= e_2 : \mu\,\tau'}\ \text{T-Bind}$$

$$\frac{\Sigma|\Gamma \vdash e : \texttt{IO}\,\tau}{\Sigma|\Gamma \vdash \texttt{spawn}\ e : \texttt{IO}\,()}\ \text{T-Spawn} \qquad \frac{\Sigma|\Gamma \vdash e : \texttt{STM}\,\tau}{\Sigma|\Gamma \vdash \texttt{atomic}\ e : \texttt{IO}\,\tau}\ \text{T-Atomic}$$

$$\frac{\Sigma|\Gamma \vdash e : \texttt{STM}\,\tau \qquad \Sigma|\Gamma \vdash e' : \texttt{STM}\,\tau \qquad \Sigma \vdash W_i \qquad \Sigma \vdash R_i \qquad \Sigma \vdash \mathcal{H}}{\Sigma|\Gamma \vdash (e, W_i, R_i, i, e', \mathcal{H}) : \texttt{IO}\,\tau}\ \text{T-Txn}$$

$$\frac{\Sigma|\Gamma \vdash e : \tau}{\Sigma|\Gamma \vdash \texttt{newref}\ e : \texttt{STM}\,(\texttt{R}\,\tau)}\ \text{T-Alloc} \qquad \frac{\Sigma|\Gamma \vdash e : \texttt{R}\,\tau}{\Sigma|\Gamma \vdash \texttt{readref}\,e : \texttt{STM}\,\tau}\ \text{T-Deref}$$

$$\frac{\Sigma|\Gamma \vdash e_1 : \texttt{R}\,\tau \qquad \Sigma|\Gamma \vdash e_2 : \tau}{\Sigma|\Gamma \vdash \texttt{writeref}\,e_1\,e_2 : \texttt{STM}\,()}\ \text{T-Assign}$$

$$\frac{\mathcal{H}(l) = (v, i) \Rightarrow \Sigma|[] \vdash v : \Sigma(l)}{\Sigma \vdash \mathcal{H}}$$

Figure 3 State related definitions.

l	$\in$ Ref	
$\mathcal{P}$	$\in$ Program	$=$ ThreadId $\rightharpoonup$ Exp
T_i	$\in$ Transaction	$=$ Exp $\times$ Store $\times$ Store $\times$ Id $\times$ Exp $\times$ Store
$\mathcal{H}, R_i, W_i$	$\in$ Store	$=$ Ref $\rightharpoonup$ Val $\times$ Id
α_i	$\in$ TxnEffect	$= \{at_i^t, ab_i^t, co_i^t(\bar{l}), r_i^{t_i}(l), \epsilon_i^t\}$
α	$\in$ Effect	$= \{\epsilon^t, sp^t(t)\}$

A transaction in progress is represented by a tuple $(e, W_i, R_i, i, e', \mathcal{H}')$. It consists of the expression e that is currently evaluated, the write set W_i and the read set R_i of the transaction, a (unique) transaction identifier i, a copy of the original transaction body e', and a copy $\mathcal{H}'$ of the heap taken at the beginning of the transaction. The latter two store the relevant state at the beginning of a transaction to facilitate the consistency check and the abort operation.

A reference corresponds to a heap location. All stores (the heap, the read set, and the write set of a transaction) map a reference to a pair of a value and a transaction identifier. The transaction identifier specifies the transaction which committed or, in case of the write set, attempts to commit the value to the global store. $S(l)$ denotes the lookup operation of a reference l in a heap S. It implies $l \in \mathrm{dom}(S)$. The store update operation $S[l \mapsto y]$ returns a store that is identical to S, except that it maps l to y. For two stores S_1 and S_2, we write $S_1[S_2]$ for the updated version of S_1 with all entries of S_2.

Operations can have different effects α on the global state: the begin transaction (at_i^t), abort transaction (ab_i^t), read reference l ($r_i^{t_i}(l)$), and commit writing references $\bar{l}$ ($co_i^t(\bar{l})$) indicating operations on the global shared heap, or empty effects (ϵ_i^t or ϵ^t), with t a thread identifier, and i a transaction id. The empty effects ϵ^t represent monadic reductions that occur outside a transaction (see top of Fig. 5). The spawn effect $sp^t(t')$denotes the spawning of a new thread with thread id t' by thread t.

The evaluation of a program with body e starts in an initial state $\langle\rangle, \{0 \mapsto e\}$ with an empty heap and a main thread with thread identifier 0. A final state has the form $\mathcal{H}, \{0 \mapsto v_0, \ldots, t_n \mapsto v_n\}$. The rules in Figures 4 and 5 define the semantics of the language constructs. In Fig. 4, $\mathcal{E}[\bullet]$ denotes an evaluation context for an expression and $\mathcal{M}[\bullet]$ an evaluation context for monadic expressions. We write m to indicate that an expression has a monadic type. As usual, $e[e'/x]$ denotes the capture-avoiding substitution of x by e' in e.

The `IO` monad is the top-level evaluation mode. Each reduction step $\stackrel{\alpha}{\Longrightarrow}$ chooses an expression from the thread pool $\mathcal{P}$. The non-determinism in this choice models an arbitrary scheduling of threads.

Spawning a thread (Spawn) creates a new entry in the thread pool and returns unit in the parent thread.

An atomic expression at the top-level (Atomic) creates a new transaction in progress with the expression to be evaluated, an empty read and write set, and a fresh transaction identifier (that has never been used before in a particular evaluation). Further, a copy

Figure 4 Operational semantics: Local evaluation steps.

Evaluation contexts

$$\begin{array}{lcl} \mathcal{E} & ::= & []\ e \mid \texttt{if}\ []\ e\ e' \\ \mathcal{M} & ::= & \texttt{readref}\ [] \mid \texttt{writeref}\ []\ e \mid []\ggg e \end{array}$$

Expression evaluation $\rightarrow$

$$\begin{array}{rcl} (\lambda x.e)\ e' & \rightarrow & e[e'/x] \\ \texttt{if tt}\ e\ e' & \rightarrow & e \\ \texttt{if ff}\ e\ e' & \rightarrow & e' \end{array}$$

$$\frac{e \rightarrow e'}{\mathcal{E}[e] \rightarrow \mathcal{E}[e']}$$

Monadic evaluation $\curvearrowright$

$$\texttt{return}\ e' \ggg e \quad \curvearrowright \quad e\ e'$$

$$\frac{e \rightarrow e'}{e \curvearrowright e'} \qquad \frac{m \curvearrowright m'}{\mathcal{M}[m] \curvearrowright \mathcal{M}[m']}$$

of the expression m is needed for possible rollbacks, and a copy of the current heap to mark the beginning of the transaction.

All monadic evaluation steps can take place inside a transaction (STM-Monad).

Allocation of a new reference (Alloc) must check that the reference is not yet allocated in the heap. But it must also check that the reference is not yet allocated in any concurrently running transaction to avoid accidental overwrites when both transactions commit. This condition is indicated by $l \notin \mathcal{H}, \mathcal{P}$, eschewing a formal definition.

Write operations (Write) are straightforward. They just affect the local write set and store the value along with the current transaction identifier.

The read operation on references (Read) needs to consult the global state. If a reference cannot be read from the local read or write set, it is accessed in the current global heap. To maintain the transaction's consistency, the read operation is successful only if the value has not been updated since the transaction's beginning. The value and transaction identifier as registered in the heap for this reference are then added to the read set and the value is returned to the transactional computation.

If a reference is present in the read set, but not in the write set, then its value is taken from the read set (ReadRSet).

If the reference is present in the write set, then its value is taken from the write set, without checking the read set (ReadWSet).

If none of the preceding three cases holds at a read, then the transaction aborts and rolls back via Rollback by abandoning the transaction in progress and reinstalling the saved transaction body m' as an atomic block. In fact, this rule has no precondition

Figure 5 Operational semantics: Global evaluation steps.

$$\frac{\mathcal{P}(t) = m \qquad m \curvearrowright m'}{\mathcal{H}, \mathcal{P} \xrightarrow{\epsilon^t} \mathcal{H}, \mathcal{P}\{t \mapsto m'\}} \text{IO-Monad}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[\texttt{spawn}\ m] \qquad t' \text{ fresh}}{\mathcal{H}, \mathcal{P} \xrightarrow{sp^t(t')} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[\texttt{return}\ ()], t' \mapsto m\}} \text{Spawn}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[\texttt{atomic}\ m] \qquad T_i = (m, \langle\rangle, \langle\rangle, i, m, \mathcal{H}) \qquad i \text{ fresh}}{\mathcal{H}, \mathcal{P} \xrightarrow{at_i^t} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[T_i]\}} \text{Atomic}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(m, W_i, R_i, i, m', \mathcal{H}')] \qquad m \curvearrowright m''}{\mathcal{H}, \mathcal{P} \xrightarrow{\epsilon_i^t} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(m'', W_i, R_i, i, m', \mathcal{H}')]\}} \text{STM-Monad}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{newref}\ e], W_i, R_i, i, m', \mathcal{H}')] \qquad l \notin \mathcal{H}, \mathcal{P}}{\mathcal{H}, \mathcal{P} \xrightarrow{\epsilon_i^t} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\mathcal{M}'[\texttt{return}\ l], W_i[l \mapsto (e, i)], R_i, i, m', \mathcal{H}')]\}} \text{Alloc}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\,\texttt{writeref}\ l\ e], W_i, R_i, i, m', \mathcal{H}')]}{\mathcal{H}, \mathcal{P} \xrightarrow{\epsilon_i^t} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\mathcal{M}'[\texttt{return}\ ()], W_i[l \mapsto (e, i)], R_i, i, m', \mathcal{H}')]\}} \text{Write}$$

$$\frac{\begin{array}{c}\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{readref}\ l], W_i, R_i, i, m', \mathcal{H}')] \\ l \notin \text{dom}(W_i) \cup \text{dom}(R_i) \qquad \mathcal{H}(l) = \mathcal{H}'(l) = (e, j)\end{array}}{\mathcal{H}, \mathcal{P} \xrightarrow{r_i^{t_i}(l)} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\mathcal{M}'[\texttt{return}\ e], W_i, R_i[l \mapsto (e, j)], i, m', \mathcal{H}')]\}} \text{Read}$$

$$\frac{\begin{array}{c}\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{readref}\ l], W_i, R_i, i, m', \mathcal{H}')] \\ l \notin \text{dom}(W_i) \qquad R_i(l) = (e, j)\end{array}}{\mathcal{H}, \mathcal{P} \xrightarrow{\epsilon_i^t} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\mathcal{M}'[\texttt{return}\ e], W_i, R_i, i, m', \mathcal{H}')]\}} \text{ReadRSet}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{readref}\ l], W_i, R_i, i, m', \mathcal{H}')] \qquad W_i(l) = (e, i)}{\mathcal{H}, \mathcal{P} \xrightarrow{\epsilon_i^t} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\mathcal{M}'[\texttt{return}\ e], W_i, R_i, i, m', \mathcal{H}')]\}} \text{ReadWSet}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(m, W_i, R_i, i, m', \mathcal{H}')]}{\mathcal{H}, \mathcal{P} \xrightarrow{ab_i^t} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[\texttt{atomic}\ m']\}} \text{Rollback}$$

$$\frac{\begin{array}{c}\mathcal{P}(t) = \mathcal{M}[(\texttt{return}\ e, W_i, R_i, i, m', \mathcal{H}')] \\ check(R_i, \mathcal{H}) = ok \qquad \mathcal{H}'' = \mathcal{H}[W_i] \qquad \bar{l} = \text{dom}(W_i)\end{array}}{\mathcal{H}, \mathcal{P} \xrightarrow{co_i^t(\bar{l})} \mathcal{H}'', \mathcal{P}[t \mapsto \mathcal{M}[\texttt{return}\ e]]} \text{Commit}$$

Figure 6 Operational semantics: Helper relations.

$$\frac{\forall l \in \mathrm{dom}(R_i) : R_i(l) = \mathcal{H}(l)}{\mathit{check}(R_i, \mathcal{H}) = \mathit{ok}}\ \textsc{Check-Ok} \qquad \frac{\exists l \in \mathrm{dom}(R_i) : R_i(l) \neq \mathcal{H}(l)}{\mathit{check}(R_i, \mathcal{H}) = \mathit{bad}}\ \textsc{Check-Bad}$$

so that a rollback may happen non-deterministically at any time during a transaction. This way, it is easy to extend our model with an explicit user abort or retry operation. Furthermore, this rule covers the abort both when reading fails as well as when the commit operation fails.

When committing (Commit), the heap is checked for updates to the references which are found in the transaction's read set since the start of the transaction. There are two cases:

The check is successful: None of the variables read by the transaction have been committed by another transaction in the meantime. Therefore, the transaction may publish its writes atomically to the shared heap and return to the IO monad.

The check fails: The only applicable rule is Rollback. The transaction aborts and restarts.

Each of these reductions generates the appropriate effect label on the transition relation. Thus, each sequence of labeled reductions uniquely determines a sequence of labels, which we call the trace of the reduction sequence. Unlike other formalizations, the interleaving of transactions as well as the abort operations are visible in the trace.

Theorem 3.1 (Type soundness)**.** *The type system in Figure 2 is sound with respect to the operational semantics of* Λ_{STM}*.*

Proof of 3.1: The proof is by establishing type preservation and progress in the usual way [9]. The proof of progress relies crucially on the use of the Rollback rule if the comparison of heap entries in Read or Commit fails. □

4 Opacity

The standard isolation property that most STM systems provide is *opacity*. It states that any allowed interleaving of transactions must have an equivalent serialized execution. Furthermore, even aborting transactions are required to view memory locations only in a consistent way.

We can prove formally that the semantics for Λ_{STM} satisfies opacity. To this end, we give a definition for well-formedness of execution traces in terms of the effects they exhibit.

We then show that reordering certain evaluation steps leads to equivalent reductions sequences. Reductions are considered equivalent if each read operation returns the same value, each commit operation commits the same values, and each transaction's outcome (abort or commit) is the same. To see which reordering yields equivalent reductions, we define a notion of dependency on effects.

Finally, we show that all reduction sequences produced by the operational semantics are equivalent to some reduction sequence with a serial trace, up to the assignment of unique labels to the transactions. Note that we only consider finite traces, without loss of generality: for infinite traces, we would be able to establish our results for all finite prefixes.

4.1 Well-formed effect traces

We start with a formal account on effect traces.

Definition 4.1 (Effect traces)**.** *A* trace $\bar{\alpha}$ *is the sequence* $\bar{\alpha} = \alpha_1 \dots \alpha_n$ *of effects* $\alpha_i \in \textbf{\textit{Effect}}, i \in 1, \dots, n$.

A total order on the effects $\alpha \in \bar{\alpha}$ *is defined by their position in the effect trace. For* $i, j \in \{1, \dots, |\bar{\alpha}|\}$ *and* $i < j$*, we use the abbreviation*

$$\bar{\alpha} \vdash \alpha_i < \alpha_j$$

to denote that an effect α_i *is happening before* α_j *in an trace* $\bar{\alpha}$*. Similarly,*

$$\bar{\alpha} \vdash \bar{\beta} < \bar{\gamma}$$

extends the relation to sets of effects if it holds pairwise for all elements in $\bar{\beta}$ *and* $\bar{\gamma}$*.*

We identify by α any effect from a trace $\bar{\alpha}$, α_i denotes the effect index i in the trace, α^t an effect from thread t, and α_i^t an effect from transaction T_i in thread t.

Further, $\bar{\alpha}|_t = \{\alpha^t \in \bar{\alpha}\}$ is the subset of all effects from thread t, and $\bar{\alpha}|_{t,i} = \{\alpha_i^t \in \bar{\alpha}\}$ the subsets of all effects from transaction i in thread t.

Empty effects encode the scheduling of threads, yet they have no influence on the globally shared state. We define therefore an operation $\langle\cdot\rangle$ which reduces a trace to its quintessence, the ordered sequence of non-empty effects.

Definition 4.2. *The kernel of a trace* $\langle\bar{\alpha}\rangle$ *is the reduction of the trace to its non-empty effects.*

$$\begin{aligned} \langle\emptyset\rangle &= \emptyset \\ \langle\alpha, \bar{\alpha}\rangle &= \begin{cases} \langle\bar{\alpha}\rangle & \text{if } \alpha = \epsilon_t^i \text{ or } \alpha = \epsilon^t \\ \alpha, \langle\bar{\alpha}\rangle & \text{otherwise} \end{cases} \end{aligned}$$

Definition 4.3. *A trace* α *is* equal in effects *to a trace* β *iff*

$$\langle\alpha\rangle \equiv \langle\beta\rangle$$

The well-formedness of a trace depends largely on the order of certain effects.

Definition 4.4 (Well-formed traces)**.** *A trace $\bar{\alpha}$ is* well-formed *iff the following conditions hold:*

- *There is no effect for a thread before its spawn effect, unless it is the main thread.*

$$\text{For } t \neq 0: \quad \alpha^t \in \bar{\alpha} \Rightarrow sp^{t'}(t) \in \bar{\alpha} \wedge \bar{\alpha} \vdash sp^{t'}(t) < \alpha^t$$

- *There is no effect for a transaction T_i before its atomic effect.*

$$r_i^{t_i}(l) \in \bar{\alpha} \Rightarrow at_i^t \in \bar{\alpha} \wedge \bar{\alpha} \vdash at_i^t < r_i^{t_i}(l)$$
$$co_i^t(\bar{l}) \in \bar{\alpha} \Rightarrow at_i^t \in \bar{\alpha} \wedge \bar{\alpha} \vdash at_i^t < co_i^t(\bar{l})$$
$$ab_i^t \in \bar{\alpha} \Rightarrow at_i^t \in \bar{\alpha} \wedge \bar{\alpha} \vdash at_i^t < ab_i^t$$

- *There is no read effect for a transaction T_i after its commit or abort effect.*

$$co_i^t(\bar{l}) \in \bar{\alpha} \Rightarrow \forall r_i^{t_i}(l) \in \bar{\alpha} : \bar{\alpha} \vdash r_i^{t_i}(l) < co_i^t(\bar{l})$$
$$ab_i^t \in \bar{\alpha} \Rightarrow \forall r_i^{t_i}(l) \in \bar{\alpha} : \bar{\alpha} \vdash r_i^{t_i}(l) < ab_t^i$$

- *A transaction may have either a commit or an abort effect, but not both.*

$$co_i^t(\bar{l}) \in \bar{\alpha} \Rightarrow ab_i^t \notin \bar{\alpha}$$
$$ab_i^t \in \bar{\alpha} \Rightarrow co_i^t(\bar{l}) \notin \bar{\alpha}$$

- *There are no non-transactional effects within a transaction.*

$$\epsilon^t \in \bar{\alpha} \Rightarrow \nexists i : \bar{\alpha} \vdash at_i^t < \epsilon^t < co_i^t(\bar{l}) \text{ or } \bar{\alpha} \vdash at_i^t < \epsilon^t < ab_i^t$$

- *Transactional effects from the same thread do not interleave.*

$$\forall t \, \forall i \neq j : \bar{\alpha} \vdash \bar{\alpha}|_{t,i} < \bar{\alpha}|_{t,j} \text{ or } \bar{\alpha} \vdash \bar{\alpha}|_{t,j} < \bar{\alpha}|_{t,i}$$

Definition 4.5 (Pending transactions)**.** *A transaction T_i is pending in a trace $\bar{\alpha}$ if it has neither a commit or an abort effect:*

$$ab_i^t \notin \bar{\alpha} \text{ and } co_i^t(\bar{l}) \notin \bar{\alpha}$$

In contrast to other definitions of well-formed execution traces (e.g. [8]), we do not include the condition that the order of all reads and writes in the transaction is preserved in the effect traces. The operational semantics guarantees that each transaction is working on a consistent view of the shared memory as indexed by its time stamp. A read operation returns the last value written, either by another transaction which updated the global heap, or by the transaction itself in a local write step. Further, all write operations are published (i.e., made visible to other transactions) only after the successful commit. Therefore, the trace reflects the order of the globally visible effects of the read and write operations. The local reads and writes have no globally visible effect.

Lemma 4.1. *All traces produced by type-correct programs are well-formed.*

Proof of 4.1: Type-correct programs allow only certain compositions of transactional phases. Effects are only produced when evaluating expressions in the `STM` monad. An at_i^t effect is only produced when entering the `STM` monad. All read effects are produced within the `STM` part, and the evaluation of a transactional expression finishes with either an ab_i^t or $co_i^t(\bar{l})$ effect. □

The well-formedness of a trace relates the effects of one transaction to each other. Complementary, an isolation level defines a relation between the effects of all transactions that participate in a trace [2]. Serializability, for example, is one of these isolation levels.

Definition 4.6 (Serial traces)**.** *A well-formed trace $\bar{\alpha}$ is serial if for any two transactions T_i and T_j $(i \neq j)$, all effects from T_i occur before all effects from T_j, or vice versa:*

$$\forall\, i \neq j : \bar{\alpha} \vdash \bar{\alpha}|_{t_i,i} < \bar{\alpha}|_{t_j,j} \text{ or } \bar{\alpha} \vdash \bar{\alpha}|_{t_j,j} < \bar{\alpha}|_{t_i,i}$$

In contrast to other approaches, we do not exclude aborting or pending transactions in the definition for serial traces. Therefore, we actually model opaque traces.

Definition 4.7 (Consistent snapshot)**.** *A transaction operates on a* consistent memory snapshot *iff there is no update of a variable between the begin of the transaction and a read effect of this variable in a transaction.*

$$\forall r_i^{t_i}(l) \in \bar{\alpha} : \nexists co_j^{t_j}(\bar{l}) \text{ with } \bar{\alpha} \vdash at_i^t < co_j^{t_j}(\bar{l}) < r_i^{t_i}(l) \text{ and } l \in \bar{l}$$

Beside the total order that is defined by the position in a trace, another partial order is connects effects based on their interdependence.

Definition 4.8 (Control dependency)**.** *An effect α_i has a* control dependency *on an effect α_j, $\alpha_i \triangleright_c \alpha_j$, iff they must occur in that order in any well-formed trace. A control dependency exists in the following cases:*

- $at_i^t \triangleright_c r_i^{t_i}(l)$
- $at_i^t \triangleright_c co_i^t(\bar{l})$
- $at_i^t \triangleright_c ab_i^t$
- $r_i^{t_i}(l) \triangleright_c ab_i^t$
- $r_i^{t_i}(l) \triangleright_c co_i^t(\bar{l})$

Definition 4.9 (Data dependency)**.** *An effect α_i has a* data dependency *on an effect α_j, $\alpha_i \triangleright_d \alpha_j$, if they exhibit a write-read, read-write or write-write conflict. A data dependency exists in the following cases where $i \neq j$:*

- $r_i^{t_i}(l) \triangleright_d co_j^{t'}(\bar{l})$ *if* $l \in \bar{l}$

- $co_i^t(\bar{l}) \triangleright_d r_j^{t'}(l)$ *if* $l \in \bar{l}$
- $co_i^t(\bar{l}) \triangleright_d co_j^{t'}(\bar{l'})$ *if* $\bar{l} \cap \bar{l'} \neq \emptyset$

Definition 4.10 (Dependency)**.** *An effect* α_i *is* directly dependent *on an effect* α_j*, iff* α_i *is either control or data dependent on* α_j *and* $\bar{\alpha} \vdash \alpha_i < \alpha_j$ *in a trace* $\bar{\alpha}$*. An effect* α_i *is* dependent *on an effect* α_j*,* $\alpha_i \triangleright \alpha_j$*, iff they are contained in the transitive closure of directly dependent effects:*

$$(\alpha_i, \alpha_j) \in \{(\alpha, \beta) \mid \alpha \text{ is directly dependent on } \beta\}^*$$

Effects that are not dependent are called independent*.*

Definition 4.11 (Dependent transactions)**.** *A transaction* T_i *is dependent on a transaction* T_j *if* $\alpha_i \triangleright \alpha_j$ *for an effect* α_i *from* T_i *and an effect* α_j *from* T_j*.*

Definition 4.12 (Trace dependencies)**.** *Let* $\bar{\alpha}$ *be a well-formed trace. The trace dependencies* $\Delta(\bar{\alpha})$ *are defined as the set of all tuples of dependent effects in this trace:*

$$\Delta(\bar{\alpha}) = \{(\alpha_i, \alpha_j) \mid \alpha_i \triangleright \alpha_j\}$$

4.2 Serializing effect traces

We are interested in equivalence classes of traces that are permutations of each other and yield the same final program state. However, there are restrictions on what reorderings of effects are permissible. For example, the order of trace items with respect to one thread must not be changed. We call permutations that leave the relative ordering inside every thread unchanged *admissible permutations*. We prove in this section that we can admissibly permute any trace from executing a program in Λ_{STM} such that it becomes serial, and that the execution of the serial trace yields the same final program state.

Definition 4.13 (Equivalence of traces)**.** *A trace* $\bar{\alpha}$ *is equivalent to a trace* $\bar{\beta}$ *iff* $\bar{\beta}$ *is an admissible permutation of* $\bar{\alpha}$ *and* $\Delta(\bar{\alpha}) = \Delta(\bar{\beta})$*.*

Definition 4.14 (Equivalence of program states)**.** *A program state* $\mathcal{P}$ *is equivalent to a program state* $\mathcal{P}'$*,* $\mathcal{P} \simeq \mathcal{P}'$ *iff for all threads* i *either* $\mathcal{P}(i) = \mathcal{P}'(i)$ *or* $\mathcal{P}(i) = \mathcal{M}[(m_1, W_i, R_i, i, m_2, \mathcal{H})], \mathcal{P}'(i) = \mathcal{M}[(m_1, W_i, R_i, i, m_2, \mathcal{H}')]$ *and* $\mathcal{H}|_{R_i} = \mathcal{H}'|_{R_i}$*.*

Definition 4.15 (Equivalence of evaluation states)**.** *An evaluation state* $\mathcal{H}, \mathcal{P}$ *is equivalent to an evaluation state* $\mathcal{H}', \mathcal{P}'$ *iff* $\mathcal{H} = \mathcal{H}'$ *and* $\mathcal{P} \simeq \mathcal{P}'$*.*

Lemma 4.2 (Permutation of reduction steps)**.** *Let* R *be the two-step reduction*

$$\mathcal{H}, \mathcal{P} \overset{\alpha_i}{\Longrightarrow} \mathcal{H}_0, \mathcal{P}_0 \overset{\alpha_j}{\Longrightarrow} \mathcal{H}', \mathcal{P}'_0.$$

If α_i *is independent from* α_j*, then there exists an equivalent reduction sequence* R' *of the form*

$$\mathcal{H}, \mathcal{P} \overset{\alpha_j}{\Longrightarrow} \mathcal{H}_1, \mathcal{P}_1 \overset{\alpha_i}{\Longrightarrow} \mathcal{H}', \mathcal{P}'_1$$

and $\mathcal{P}'_0 \sim \mathcal{P}'_1$*.*

Proof of 4.2: Case distinction on all independent effects.

Effect-free operations ($\alpha_i = \epsilon^t$ or $\alpha_j = \epsilon_i^t$) are either pure or work on local (transactional) state. Therefore, these steps can get swapped with any operation while resulting in the same heap and thread pool. Reduction steps which result in an abort only modify the transactions' local state. The same holds for read operations. For commit effects, it holds that subsequent independent commit operations change disjoint parts of the global heap. The rules for inconsistency checks require that read and write sets of concurrently running transactions are disjoint in case of successful commit. Therefore reordering independent commit operations has no influence on the heap's final state, and produces equivalent program states. □

In our semantics the begin of a transaction defines its relative order to other transactions. Yet, this order is only partial for transactions that perform their operations interleaved. In this case, they all only commit successfully if their operations do not conflict with each other. The following lemma shows that for these transactions, any relative order is permissive.

Lemma 4.3 (Permutation of committing transactions)**.** *Let $\bar{\alpha}$ be a well-formed trace with $\bar{\alpha} = \bar{\alpha}' co_i^t(\bar{l})$ and $\bar{\alpha}'$ serial. Further, let T_j be a transaction with $at_j^{t_j} \in \bar{\alpha}$ and $\bar{\alpha} \vdash at_i^{t_i} < at_j^{t_j} < co_i^{t_i}(\bar{l})$ and there does not exist a k with $\bar{\alpha} \vdash at_i^{t_i} < at_k^{t_k} < at_j^{t_j}$. Then $\bar{\alpha}$ is equivalent to a trace $\bar{\beta}$ with $\bar{\beta} \vdash \alpha_j < at_i^{t_i}$ for all effects α_j of transaction T_j.*

Proof of 4.3: According to the restrictions, the trace must have the following structure:

$$\bar{\alpha} = \alpha_{pre}, at_i^{t_i}, \overline{r_i^{t_i}(l) \mid \epsilon_i^{t_i}}, at_j^{t_j}, \overline{r_j^{t_j}(l) \mid \epsilon_j^{t_j}}, (ab_i^{t_i} \mid co_i^{t_i}(\bar{l})), \alpha_{post}$$

There are no dependencies between $at_j^{t_j}$ and any $r_i^{t_i}(l)$, or $at_j^{t_j}$ and $at_i^{t_i}$, or any $r_i^{t_i}(l)$ and any $r_j^{t_j}(l)$. Empty effects neither introduce dependencies. By Lemma 4.3 this is therefore equivalent to trace

$$\alpha_{pre}, at_j^{t_j}, \overline{r_j^{t_j}(l) \mid \epsilon_j^{t_j}}, at_i^{t_i}, \overline{r_i^{t_i}(l) \mid \epsilon_i^{t_i}}, (ab_i^{t_i} | co_i^{t_i}(\bar{l})), \alpha_{post}$$

Case distinction on the status of T_j.

- *Case* $ab_j^{t_j} \in \bar{\alpha}$: There is no dependency between $ab_j^{t_j}$ and any effect of T_i, so by Lemma 4.2, the trace is equivalent to

$$\alpha_{pre}, at_j^{t_j}, \overline{r_j^{t_j}(l) \mid \epsilon_j^{t_j}}, ab_j^{t_j}, at_i^{t_i}, \overline{r_i^{t_i}(l) \mid \epsilon_i^{t_i}}, \alpha_{post}$$

- *Case* $co_j^{t_j}(\bar{l}) \in \bar{\alpha}$: Assume that $r_i^{t_i}(l) \triangleright co_j^{t_j}(\bar{l})$. Then, the validation of the transaction T_i in rule Commit would fail and $co_i^{t_i}(\bar{l}) \notin \bar{\alpha}$ in contradiction to the assumption. Hence, $co_j^{t_j}(\bar{l})$ is not dependent on any effect of T_i, and by Lemma 4.2, the trace is equivalent to

$$\alpha_{pre}, at_j^{t_j}, \overline{r_j^{t_j}(l) \mid \epsilon_j^{t_j}}, co_j^{t_j}(\bar{l}), at_i^{t_i}, \overline{r_i^{t_i}(l) \mid \epsilon_i^{t_i}}, \alpha_{post}$$

- *Case* T_j is pending: Then the trace $\bar{\alpha}$ is equivalent to

$$\alpha_{pre}, at_j^{t_j}, \overline{r_j^{t_j}(l) \mid \epsilon_j^{t_j}}, at_i^{t_i}, \overline{r_i^{t_i}(l) \mid \epsilon_i^{t_i}}, \alpha_{post}$$

End case distinction on the status of T_j. □

In the remainder of this section, we identify which subsequences of a trace are not serial, and specify an algorithm that moves the effects to the appropriate place.

Lemma 4.4 (Conflicts)**.** *Let $\bar{\alpha}$ be a well-formed trace. Then $\bar{\alpha}$ is either serial, or there exists an α_k such that the prefix $\alpha_1 \dots \alpha_k$ is serial and*

1. *α_k and α_{k+1} are independent, or*
2. *$\alpha_k = r_i^{t_i}(l)$ and $\alpha_{k+1} = co_j^{t_j}(\bar{l})$ with $l \in \bar{l}$.*

Proof of 4.4: We consider all possible combinations of effects which might occur in a well-formed trace. Cases that are left out lead violate well-formedness.
Case distinction on α_i and α_{k+1} where $i \neq j$.

- *Case* $\alpha_k = \epsilon^{t_i}$ or $\alpha_{k+1} = \epsilon^{t_j}$: serial or independent.
- *Case* $\alpha_k = \epsilon_j^{t_i}$ or $\alpha_{k+1} = \epsilon_j^{t_j}$: serial or independent.
- *Case* $\alpha_k = at_i^{t_i}$ and $\alpha_{k+1} = at_j^{t_j}$: serial.
- *Case* $\alpha_k = at_i^{t_i}$ and $\alpha_{k+1} = r_i^{t_i}(l)$: serial.
- *Case* $\alpha_k = at_i^{t_i}$ and $\alpha_{k+1} = r_j^{t_j}(l)$: independent.
- *Case* $\alpha_k = at_i^{t_i}$ and $\alpha_{k+1} = co_i^{t_i}(\bar{l})$: serial.
- *Case* $\alpha_k = at_i^{t_i}$ and $\alpha_{k+1} = co_j^{t_j}(\bar{l})$: independent.
- *Case* $\alpha_k = at_i^{t_i}$ and $\alpha_{k+1} = ab_i^{t_i}$: serial.
- *Case* $\alpha_k = at_i^{t_i}$ and $\alpha_{k+1} = ab_j^{t_j}$: independent.
- *Case* $\alpha_k = r_i^{t_i}(l)$ and $\alpha_{k+1} = r_j^{t_j}(l')$: independent.
- *Case* $\alpha_k = r_i^{t_i}(l)$ and $\alpha_{k+1} = r_i^{t_i}(l')$: serial.
- *Case* $\alpha_k = r_i^{t_i}(l)$ and $\alpha_{k+1} = co_i^{t_i}(\bar{l})$: serial.
- *Case* $\alpha_k = r_i^{t_i}(l)$ and $\alpha_{k+1} = co_j^{t_j}(\bar{l})$: If $l \in \bar{l}$, then this is the second case in the lemma. Otherwise independent.
- *Case* $\alpha_k = r_i^{t_i}(l)$ and $\alpha_{k+1} = ab_i^{t_i}$: serial.

Figure 7 Reordering transactions for opacity.

while $\bar{\alpha}$ is not serial **do**
  choose α_k and α_{k+1} such that $\alpha_1 \dots \alpha_k$ is serial and $\alpha_1 \dots \alpha_{k+1}$ is not serial
  if α_k and α_{k+1} are independent **then** swap α_k with α_{k+1}
  else if $\alpha_k = r_i^{t_i}(l)$ and $\alpha_{k+1} = co_j^{t_j}(\bar{l})$ with $l \in \bar{l}$ **then**
    permute transactions in prefix such that prefix ends with transaction j
  else
    report error
  end if
end while

- *Case* $\alpha_k = r_i^{t_i}(l)$ and $\alpha_{k+1} = ab_j^{t_j}$: independent.
- *Case* $\alpha_k = co_i^{t_i}(\bar{l})$ and $\alpha_{k+1} = co_j^{t_j}(\bar{l}')$: According to the operational semantics, it must hold that $\bar{l} \cap \bar{l}' = \emptyset$. Therefore, the effects are independent.
- *Case* $\alpha_k = co_i^{t_i}(\bar{l})$ and $\alpha_{k+1} = ab_j^{t_j}$: independent.

End case distinction on α_i and α_{k+1} where $i \neq j$. □

For the proof of opacity, we define an algorithm which produces for a serializable trace an equivalent serial trace.

The algorithm in Figure 7 has the following properties:

1. It terminates on all traces produced by a type-correct programs in Λ_{STM} without an error.
2. For any trace input of a type-correct program in Λ_{STM}, it yields an equivalent serial trace.

We prove these properties in several steps.

Lemma 4.5 (Termination)**.** *The algorithm terminates on all traces of well-typed programs in* Λ_{STM}*.*

Proof of 4.5: We show that the algorithm only performs a finite number of swaps for each pair of effects. Further, for each iteration of the while loop, either a permutation or a swap is performed. The transaction permutations are performed at most $n!$ times, where n is the number of all transactions that participates in a trace. Let m_t denote the number of effects that a transaction produces, and $m = \max m_t$. For every permutation, each pair of effects is swapped at most once $((m-1)!)$. As each trace consists only of a finite number of effects, the algorithm performs at most $n!(m-1)!$ many swap operations. □

Lemma 4.6 (Permutation)**.** *The output of the algorithm is a permutation of the input trace.*

Proof of 4.6: All operations on the trace are permutations of effects. Therefore, effects are neither removed from nor added to the input trace. □

Lemma 4.7 (Dependencies)**.** *The algorithm does not change any dependencies in the trace.*

Proof of 4.7: Effects are only swapped when they are independent or when permuting transactions. In the latter case, the dependencies in the trace are respected as is shown in Lemma 4.3. □

Lemma 4.8 (Correctness of the algorithm)**.** *The output of the algorithm is an equivalent serial trace.*

Proof of 4.8: By 4.6 and 4.7, the output is equivalent to the input trace. By 4.5, the algorithm terminates on all traces from type-correct programs. In this case, the condition for entering the while loop is falsified, and therefore the trace is serial. □

Theorem 4.1 (Opacity)**.** *Let $\mathcal{P}_0$ be a type-correct program. Further, let R be a sequence of reductions*

$$\mathcal{H}_0, \mathcal{P}_0 \overset{\alpha_1}{\Longrightarrow} \ldots \overset{\alpha_n}{\Longrightarrow} \mathcal{H}_n, \mathcal{P}_n.$$

Then there exists an equivalent sequence R' of the form

$$\mathcal{H}_0, \mathcal{P}_0 \overset{\alpha'_1}{\Longrightarrow} \ldots \overset{\alpha'_n}{\Longrightarrow} \mathcal{H}_n, \mathcal{P}'_n$$

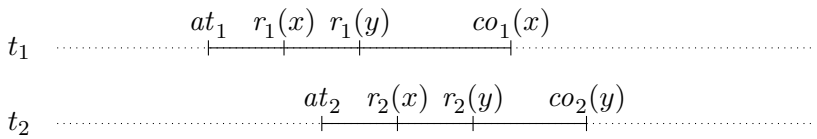
such that $\bar{\alpha}(R')$ is serial.

Proof of 4.1: We apply the algorithm for serialization of traces to the traces of R. Because the algorithm only requires the permutation of independent effects, by Lemma 4.2 the result is an equivalent reduction sequence with serial trace. □

5 Snapshot Isolation

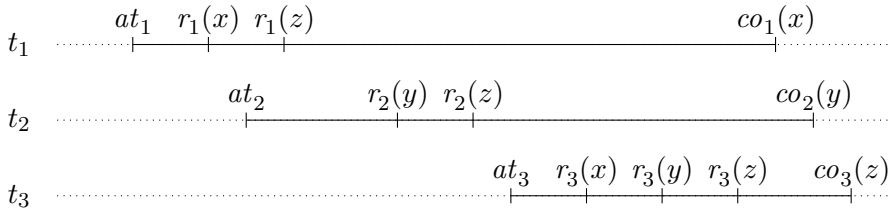
The semantics of serializability is easy to reason about, but it is rather restrictive with respect to the set of valid schedules. By confining the check for intermediate updates at commit time to the write set of the transaction, the system exhibits *snapshot isolation* semantics, a popular concurrency control notion in data bases [2]. The basic idea behind snapshot isolation is that each transaction is guaranteed to work on a consistent memory snapshot in isolation from each other, and that there are no lost updates. In the context of STM, snapshot isolation can be used to implement data structures where operations are checked for conflicts on a higher level. A typical use case are container data structures like lists or trees where insertions of different elements commute on the level of semantics, but the implementation yields non-commuting memory access patterns.

Serializable traces are trivially also valid in snapshot isolation as read and write effects of transactions are not interleaved. For further examples of snapshot traces consider the following execution trace:



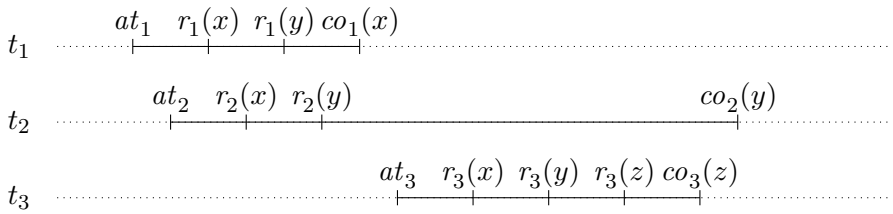
Transactions T_1, running in thread t_1 and transaction T_2, running in thread t_2, operate on the same memory snapshot, and update different memory locations. Yet, there is a read-write dependency from T_1 on T_2 because T_1 is committing x which T_2 read. Vice versa, there is also a read-write dependency from T_2 on T_1 due to their operations on y. Therefore, the transactions cannot be serialized by re-ordering their traces. This mutual read-write dependency on transactions is known as write skew anomaly in the literature [2].

To detect such a kind of anomaly, it does not suffice to consider a pair of transactions in isolation. In the following example, three transactions operate on the same memory snapshot, again committing on different locations.



There is no write skew between transactions T_1 and T_2. However, these transactions are related via their read-read dependency on variable z which is updated in transaction T_3. For this reason, T_1 and T_2 cannot be serialized with respect to each other. To reflect this read-read dependency in combination with write-skew, the snapshot relationship must contain the transitive closure of transactions that exhibit a write skew anomaly.

Further, this transitive closure needs further to be extended with transactions that only partly share data dependencies that are not write skews. The situation is depicted in this example:



Transaction T_3 can neither be ordered after T_2 because it read variable y which is updated by T_2, nor may it be ordered before T_2 due to its write-read dependency on transaction T_1. The underlying reason for this is given in the write skew of T_1 and T_2. We therefore say that T_2 and T_3 are also snapshot related.

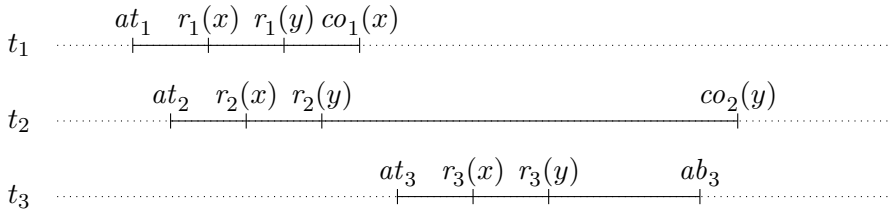
By definition, only transaction that commit successfully can exhibit write skew anomalies. For opaque systems, aborting transactions that operate on the same memory snap-

Figure 8 Operational semantics: Check for Snapshot Isolation.

$$\frac{\forall l \in \mathrm{dom}(R_i) \cap \mathrm{dom}(W_i) : R_i(l) = \mathcal{H}(l)}{check(R_i, W_i, \mathcal{H}) = ok}\ \textsc{Check-Ok}$$

$$\frac{\exists l \in \mathrm{dom}(R_i) \cap \mathrm{dom}(W_i) : R_i(l) \neq \mathcal{H}(l)}{check(R_i, W_i d, \mathcal{H}) = bad}\ \textsc{Check-Bad}$$

shot as another transaction can be ordered before this transaction. The situation changes when the aborting transaction operates on a different snapshot as this does not allow serializing it the committing transaction. Consider the following trace:



As in the previous example, it is neither possible to move transaction T_3 before nor after T_2. This indicates that both committing and aborting transactions may be snapshot related.

5.1 Operational semantics

Figure 8 shows an alternative implementation of Check-Ok and Check-Bad. Replacing the original relations in Figure 6 with these, the algorithm implements snapshot isolation. We call the language whose semantics is defined by the adapted rules in the following Λ_{SI} to distinguish it from its opaque counterpart, Λ_{STM}.

Notice that an alternative semantics of the read access to transactional variables for snapshot isolation might return the entry in the transaction local heap copy without checking for the current value in the heap. This construction also provides a consistent memory snapshot for transactional execution. In practice, this can be achieved for example by keeping multiple versions of each variable. To keep the formalizations of the different isolation levels comparable, we refrain here from modeling such a multi-versioning scheme.

5.2 Snapshot isolation for Λ_{SI}

We now formally define the notion of snapshot isolation in terms of traces. Following the definition for the isolation level of snapshot isolation in [2], none of the anomalies but write skews are allowed. Our formalization of traces impedes by design dirty reads and dirty writes because updates are only visible to other transactions after a commit.

The definition of effects does not distinguish between writes and commits, but merges them together into one effect. We therefore can focus here on the remaining kinds of anomalies.

Definition 5.1 (Non-repeatable reads)**.** *A transaction T_i experiences a non-repeatable read iff*

$$\exists x : \bar{\alpha} \vdash r_i^{t_i}(x) < co_j^{t_j}(\bar{l}) < r_i^{t_i}(x) \text{ with } x \in \bar{l}$$

Definition 5.2 (Read skew)**.** *A transaction T_i exhibits a read skew with some other transaction T_j $(i \neq j)$ iff*

$$\exists x, y : \bar{\alpha} \vdash r_i^{t_i}(x) < co_j^{t_j}(\bar{l}) < r_i^{t_i}(y) \text{ with } x, y \in \bar{l}$$

Definition 5.3 (Lost updates)**.** *A transaction causing a lost update iff*

$$\exists x : \bar{\alpha} \vdash r_i^{t_i}(x) < co_j^{t_j}(\bar{l}) < co_i^{t_i}(\bar{l}') \text{ and } x \in \bar{l} \cap \bar{l}'$$

Definition 5.4 (Snapshot isolation)**.** *A transactional system implements* snapshot isolation *iff in its trace*

1. *all reads are repeatable,*
2. *there are is no read skew, and*
3. *there are no lost updates.*

Lemma 5.1. *All reads in Λ_{SI} are repeatable.*

Proof of 5.1: In Λ_{SI}, each read value is registered in the read set for a location after the first global read access (Read). All subsequent read access either retrieve the value from the read set (ReadRSet) or the write set (ReadWSet). So, any effect trace contains at most one read effect for a location in a transaction, and non-repeatable reads are not possible. □

Lemma 5.2. *There is no read skew in Λ_{SI}.*

Proof of 5.2: In Read, there is a test for intermediate updates to the heap:

$$\mathcal{H}(l) = \mathcal{H}'(l) = (e, j)$$

The value in the global heap can only have changed when another transaction T_j committed to the location. Therefore, the global read only succeeds if there is no read skew possible. □

Lemma 5.3. *There are no lost updates in Λ_{SI}.*

Proof of 5.3: In COMMIT, there is a check for intermediate updates to the heap:

$$\forall l \in \mathrm{dom}(R_i) \cap \mathrm{dom}(W_i) : R_i(l) = \mathcal{H}(l)$$

This check fails if another transaction T_j committed to the location between the global read of the value that added it to the read set and the commit. Therefore, the commit only succeeds if there are no lost updates possible. □

Theorem 5.1 (Snapshot isolation for Λ_{SI})**.** *The formal system Λ_{SI} implements snapshot isolation.*

Proof of 5.1: The proof follows immediately from Lemmas 5.1,5.2, and 5.3. □

5.3 Snapshot traces

In Section 4, we have shown how to transform serializable traces into a canonical form, namely serial traces. For traces from systems with snapshot isolation, we now introduce also a canonical form which we call snapshot trace.

The main difference between serializability (or opacity) and snapshot isolation is the presence of write skews. Starting from their definition, we derive several criteria that prevent the serialization of a snapshot trace.

Definition 5.5 (Write skew anomaly)**.** *Two transactions T_i and T_j, $i \neq j$, exhibit a write skew anomaly in a trace $\bar{\alpha}$, $T_i \sim_{\bar{\alpha}} T_j$, iff there exists references x, y such that*

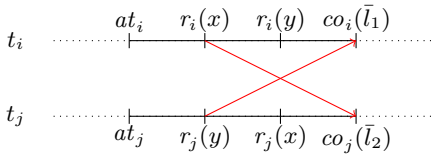
$$r_i^{t_i}(x), r_i^{t_i}(y), r_j^{t_j}(x), r_j^{t_j}(y), co_i^{t_i}(\bar{l}_1), co_j^{t_j}(\bar{l}_2) \in \bar{\alpha}$$

and

$$\bar{\alpha} \vdash at_i^{t_i} < co_j^{t_j}(\bar{l}_2)$$

$$\bar{\alpha} \vdash at_j^{t_j} < co_i^{t_i}(\bar{l}_1)$$

with $x \in \bar{l}_1$ and $y \in \bar{l}_2$.



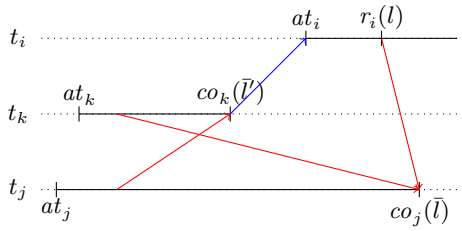
Lemma 5.4. *The write skew anomaly $\sim_{\bar{\alpha}}$ defines a symmetric relation on the set of transactions in a trace $\bar{\alpha}$.*

Proof. The symmetry follows directly from Definition 5.5. □

Let $\sim_{\bar{\alpha}}^{+}$ be the transitive and reflexive closure of $\sim_{\bar{\alpha}}$.

Definition 5.6 (Snapshot related)**.** *A transaction T_i is snapshot related with another transaction T_j, $T_i \ltimes T_j$, iff*

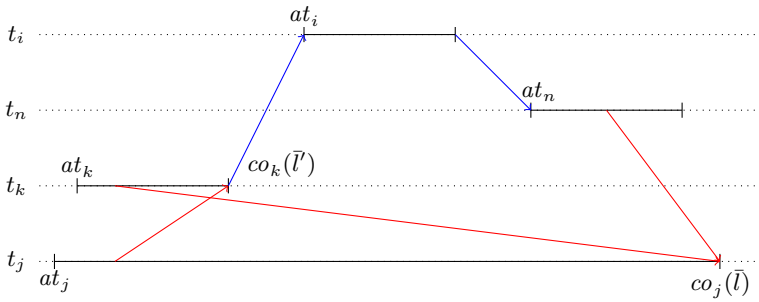
1. $\bar{\alpha} \vdash at_j^{t_j} < at_i^{t_i} < co_j^{t_j}(\bar{l})$,
2. *there is* $r_i^{t_i}(l)$ *such that* $\bar{\alpha} \vdash r_i^{t_i}(l) < co_j^{t_j}(\bar{l})$ *with* $l \in \bar{l}$,
3. *there is a* T_k *such that* T_i *depends on* T_k *and* $T_k \sim_{\bar{\alpha}}^{+} T_j$.



Obviously, snapshot dependency is not a symmetric relationship. It further holds that if there is a write skew between T_i and T_j, then $T_i \ltimes T_j$ or vice versa.

Definition 5.7 (Snapshot connected)**.** *A transaction T_i is snapshot connected to another transaction T_j,$T_i \backsimeq T_j$, iff*

1. $\bar{\alpha} \vdash at_j^{t_j} < at_i^{t_i} < co_j^{t_j}(\bar{l})$,
2. *there is a* T_k *on which* T_i *depends with* $T_k \sim_{\bar{\alpha}}^{+} T_j$, *and*
3. *there is a* T_n *such that* T_i *depends on* T_n *and* $T_n \ltimes T_j$



Lemma 5.5. *Transactions that are snapshot connected cannot be serialized.*

Proof of 5.5: We consider the two possible cases:
Case distinction .

- *Case* Ordering T_i before T_j: By definition, there is a transaction T_k with $T_k \sim_{\bar{\alpha}}^{+} T_j$, and T_i depends on T_k. T_k cannot be ordered before T_j because there is a read effect $r_k^{t_k}(l)$ which is data dependent on the commit effect $co_j^{t}(\bar{l})$. Due to the dependency on T_i must be ordered after T_k.

- *Case* Ordering T_i before T_j: By definition, there is transaction T_n with $T_n \ltimes T_j$, and T_n depends on T_i. Due to a read effect $r_n^{t_n}(l)$, T_n cannot be moved after the commit effect $co_j^t(\bar{l})$ because the read effect $l \in \bar{l}$ is data dependent on it. Further, transaction T_i may not be ordered after T_n because of T_n depends on T_i.

End case distinction . □

Definition 5.8 (Snapshot admissible)**.** *A well-formed trace $\bar{\alpha}$ is snapshot admissible if*

1. *the atomic effect of a transaction is followed by all read effects and effects of this transaction,*
2. *there are no lost updates, and*
3. *if there is an atomic effect $at_i^{t_i}$ such that $\bar{\alpha} \vdash at_j^{t_j} < at_i^{t_i} < co_j^{t_j}(\bar{l})$, then T_i is snapshot connected to T_j, and all effects from T_i are before $co_j^{t_j}(\bar{l})$.*

Lemma 5.6. *Snapshot admissible traces are snapshot traces.*

Proof of 5.6: By definition, snapshot admissible traces do not allow lost updates. Further, it holds that no effect from another transaction is allowed between the reads of a transaction. Hence, neither read skew nor non-repeatable reads are possible. □

Examples Trivially, serialized traces are snapshot admissible as there is no interleaving of transactional effects from different threads possible.

Corollary 5.1. *A serialized trace is snapshot admissible.*

For traces with write skew and snapshot-connected transactions, the canonical snapshot admissible trace are of the following form:

$$at_i^{t_i}\ \overline{(r_i^{t_i}(l) \mid \epsilon_i^{t_i})}\ at_j^{t_j}\ \overline{(r_j^{t_j}(l') \mid \epsilon_j^{t_j})}\ co_i^{t_j}(\bar{l})\ co_j^{t_j}(\bar{l'})$$
$$at_i^{t_i}\ \overline{(r_i^{t_i}(l) \mid \epsilon_i^{t_i})}\ at_j^{t_j}\ \overline{(r_j^{t_j}(l') \mid \epsilon_j^{t_j})}\ co_i^{t_j}(\bar{l})\ at_k^{t_k}\ \overline{(r_k^{t_k}(l'') \mid \epsilon_k^{t_k})}\ ab_k^{t_k}\ co_j^{t_j}(\bar{l'})$$

To show that traces produced by well-typed programs in Λ_{SI} are snapshot admissible, we follow a similar path as in the proof for opaqueness of Λ_{STM}. As for opaque traces, it is possible to reorder traces under certain conditions without changing their semantics.

To simplify the reordering, we collect snapshot connected transactions in a special relation $\fallingdotseq$ as we encounter them in the process.

Definition 5.9 (Snapshot admissible)**.** *Given a relation $\fallingdotseq$ between transactions, a well-formed trace $\bar{\alpha}$ is snapshot admissible under $\fallingdotseq$ iff*

1. *$\bar{\alpha}$ is snapshot admissible, and*
2. *if transaction T_i is snapshot connected with T_j, then $T_i \fallingdotseq^* T_j$.*

Lemma 5.7 (Conflicts)**.** *A trace $\bar{\alpha}$ taken from an execution of a program in Λ_{SI} is either snapshot admissible under some $\div$, or there is a prefix $\bar{\alpha}'$ such that $\bar{\alpha}' = \alpha_1 \dots \alpha_k$ is snapshot admissible under $\div$ and*

- *α_k and α_{k+1} are independent, or*
- *$\alpha_k = r_i^{t_i}(l)$ and $\alpha_{k+1} = co_j^t(\bar{l})$ with $l \in \bar{l}$.*

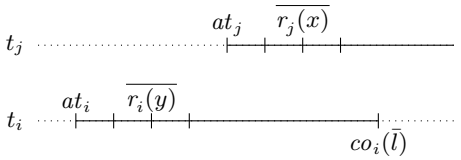
Proof of 5.7: We consider all possible combinations of effects which might occur in a well-formed trace in Λ_{SI}. Cases that are left out violate well-formedness.
Case distinction on α_i and α_{k+1} where $i \neq j$.

- *Case* $\alpha_k = \epsilon^{t_i}$ or $\alpha_{k+1} = \epsilon^{t_j}$: snapshot admissible or independent.
- *Case* $\alpha_k = \epsilon_j^{t_i}$ or $\alpha_{k+1} = \epsilon_j^{t_j}$: snapshot admissible or independent.
- *Case* $\alpha_k = at_i^{t_i}$ and $\alpha_{k+1} = at_j^{t_j}$: snapshot admissible.
- *Case* $\alpha_k = at_i^{t_i}$ and $\alpha_{k+1} = r_i^{t_i}(l)$: snapshot admissible.
- *Case* $\alpha_k = at_i^{t_i}$ and $\alpha_{k+1} = r_j^{t_j}(l)$: independent.
- *Case* $\alpha_k = at_i^{t_i}$ and $\alpha_{k+1} = co_i^{t_i}(\bar{l})$: snapshot admissible.
- *Case* $\alpha_k = at_i^{t_i}$ and $\alpha_{k+1} = co_j^{t_j}(\bar{l})$: independent.
- *Case* $\alpha_k = at_i^{t_i}$ and $\alpha_{k+1} = ab_i^{t_i}$: snapshot admissible.
- *Case* $\alpha_k = at_i^{t_i}$ and $\alpha_{k+1} = ab_j^{t_j}$: independent.
- *Case* $\alpha_k = r_i^{t_i}(l)$ and $\alpha_{k+1} = r_j^{t_j}(l')$: independent.
- *Case* $\alpha_k = r_i^{t_i}(l)$ and $\alpha_{k+1} = r_i^{t_i}(l')$: snapshot admissible.
- *Case* $\alpha_k = r_i^{t_i}(l)$ and $\alpha_{k+1} = co_i^{t_i}(\bar{l})$: snapshot admissible.
- *Case* $\alpha_k = r_i^{t_i}(l)$ and $\alpha_{k+1} = co_j^{t_j}(\bar{l})$: If $l \in \bar{l}$, then this is the second case in the lemma. Otherwise independent.
- *Case* $\alpha_k = r_i^{t_i}(l)$ and $\alpha_{k+1} = ab_i^{t_i}$: snapshot admissible.
- *Case* $\alpha_k = r_i^{t_i}(l)$ and $\alpha_{k+1} = ab_j^{t_j}$: independent.
- *Case* $\alpha_k = co_i^{t_i}(\bar{l})$ and $\alpha_{k+1} = co_j^{t_j}(\bar{l}')$: According to the operational semantics, it must hold that $\bar{l} \cap \bar{l}' = \emptyset$. Therefore, the effects are independent.
- *Case* $\alpha_k = co_i^{t_i}(\bar{l})$ and $\alpha_{k+1} = ab_j^{t_j}$: independent.

End case distinction on α_i and α_{k+1} where $i \neq j$. □

Lemma 5.8 (Permutation of independent transactions)**.** *Let $\bar{\alpha}$ be a well-formed trace with $\bar{\alpha} = \bar{\alpha}' co_i^t(\bar{l})$ and $\bar{\alpha}'$ is snapshot admissible. Further, let T_j be a transaction with $\bar{\alpha} \vdash at_i^{t_i} < at_j^{t_j} < co_i^{t_i}(\bar{l})$ and there $\nexists k$ with $\bar{\alpha} \vdash at_i^{t_i} < at_k^{t_k} < at_j^{t_j}$.*

Then, either T_i and T_j have a write skew, or trace $\bar{\alpha}$ is equivalent to a trace $\bar{\beta}$ with $\bar{\beta} \vdash \alpha_j < at_i^{t_i}$ for all effects α_j of transaction T_j.



Proof of 5.8: There are no dependencies between $at_j^{t_j}$ and any $r_i^{t_i}(l)$, or $at_j^{t_j}$ and $at_i^{t_i}$, or any $r_i^{t_i}(l)$ and any $r_j^{t_j}(l)$.

If $co_j^t(\bar{l}) \in \bar{\alpha}$ and there is a dependency of a $r_i^{t_i}(l)$ to $co_j^t(\bar{l})$, then the transactions have a write skew. Otherwise, T_j does not depend on T_i and all effects may be reordered because they are independent. □

Examples To provide some intuition on the permutations in Algorithm 9, we consider the steps that are performed on the following trace.

$$at_1\ at_2\ r_1(x)\ r_1(y)\ r_2(x)\ r_2(y)\ at_3\ r_3(z)\ at_4\ r_4(z)\ r_4(x)\ co_1(x)\ co_2(y)\ ab_4\ co_3(z)$$

For better readability, we assume that each transaction is run in a separate thread.

Starting with an empty $\div$, the algorithm detects the prefix $at_1\ at_2\ r_1(x)$ to not be admissible and reorders the independent effects at_2 and $r_1(x)$. Similarly, the other read effects of T_1 are moved towards at_1, yielding

$$at_1\ r_1(x)\ r_1(y)\ at_2\ r_2(x)\ r_2(y)\ at_3\ r_3(z)\ at_4\ r_4(z)\ r_4(x)\ co_1(x)\ co_2(y)\ ab_4\ co_3(z)$$

The admissible prefix ends here with $r_4(x)$ which is followed by $co_1(x)$. The atomic effect between at_4 and at_1 which is closest to at_4 is at_3. Because T_4 does not depend on T_3, the effect at_4 is moved before at_3. With the following recursive calls to TxnSwap, at_4 finally ends up at the head of the trace.

$$at_4\ at_1\ r_1(x)\ r_1(y)\ at_2\ r_2(x)\ r_2(y)\ at_3\ r_3(z)\ r_4(z)\ r_4(x)\ co_1(x)\ co_2(y)\ ab_4\ co_3(z)$$

The next steps move again the read effects of T_4 immediately behind at_4, such that the trace is then in this order:

$$at_4\ r_4(z)\ r_4(x)\ at_1\ r_1(x)\ r_1(y)\ at_2\ r_2(x)\ r_2(y)\ at_3\ r_3(z)\ co_1(x)\ co_2(y)\ ab_4\ co_3(z)$$

Figure 9 Reordering transactions for snapshot traces.

$\fallingdotseq = \emptyset$
while $\bar{\alpha}$ is not snapshot admissible under $\fallingdotseq$ **do**
choose α_k and α_{k+1} such that $\alpha_1 \dots \alpha_k$ is snapshot admissible under $\fallingdotseq$ and $\alpha_1 \dots \alpha_{k+1}$ is not
if α_k is not in conflict with α_{k+1} **then**
swap α_k with α_{k+1}
else if $\alpha_k = r_i^{t_i}(l)$ and $\alpha_{k+1} = co_j^{t_j}(\bar{l})$ **then**
TxnSwap(i,j)
else
signal error
end if
end while

method TxnSwap(i,j)
if $\nexists at_k^{t_k} : \bar{\alpha} \vdash at_j^{t_j} < at_k^{t_k} < at_i^{t_i}$ **then**
if write skew between T_i and T_j and not $T_i \fallingdotseq T_j$ **then**
add (T_i, T_j) to $\fallingdotseq$
else
move $at_i^{t_i}$ before $at_j^{t_j}$
end if
else
take $at_n^{t_n}$ such that $\nexists at_k^{t_k} \bar{\alpha} \vdash at_n^{t_n} < at_k^{t_k} < at_i^{t_i}$
if T_i depends on T_n **then**
if $T_n \fallingdotseq T_j$ **then**
add (T_i, T_j) to $\fallingdotseq$
else
TxnSwap(n,j)
end if
else
move $at_i^{t_i}$ before $at_n^{t_n}$
TxnSwap(i,j)
end if
end if
end

Now, the commit of T_1 is permuted with the preceding effects resulting in the trace

$$at_4\ r_4(z)\ r_4(x)\ at_1\ r_1(x)\ r_1(y)\ at_2\ r_2(x)\ co_1(x)\ r_2(y)\ at_3\ r_3(z)\ co_2(y)\ ab_4\ co_3(z)$$

with admissible prefix $at_4\ r_4(z)\ r_4(x)\ at_1\ r_1(x)\ r_1(y)\ at_2\ r_2(x)$. At this point, the algorithm detects the write skew between T_1 and T_2 and adds $(1, 2)$ to $\div$. The prefix $at_4\ r_4(z)\ r_4(x)\ at_1\ r_1(x)\ r_1(y)\ at_2\ r_2(x)\ co_1(x)$ is admissible under the extended $\div$, and in the next iteration $r_2(y)$ is again moved before $co_1(x)$.

Finally, swapping the commits and aborts at the end of the trace towards the other effects of the transaction to which they belong, the trace is snapshot admissible:

$$at_4\ r_4(z)\ r_4(x)\ ab_4\ at_1\ r_1(x)\ r_1(y)\ at_2\ r_2(x)\ r_2(y)\ co_1(x)\ at_3\ r_3(z)\ co_3(z)$$

The algorithm in Figure 9 has the following properties:

1. It terminates on all traces that are produced by well-typed programs in Λ_{SI}.
2. For any trace input from a well-typed program in Λ_{SI}, it yields an equivalent trace which is snapshot admissible.

Lemma 5.9 (Termination)**.** *The algorithm terminates on all traces of type-correct programs in Λ_{SI} without an error.*

Proof of 5.9: Let n be the number of transaction in the trace. Each permutation of atomic effects is performed at most $n - 1$ times. For every permutation, each effect is swapped at most (m) times where m is the maximal number of effects which are produced by a transaction in the trace. Hence, the algorithm stops after a finite number of swaps.

By Lemma 5.7, the program never reaches the case for signaling the error. □

Lemma 5.10 (Permutation)**.** *The output of the algorithm is a permutation of the input trace.*

Proof of 5.10: All operations on the trace only permute the effects, but do not change the elements in the trace. □

Lemma 5.11 (Dependencies)**.** *The algorithm does not change any dependencies in the trace.*

Proof of 5.11: Effects are only swapped when they are independent or when permuting transactions. In the latter case, the dependencies in the trace are respected as is shown in Lemma 5.8. □

Theorem 5.2 (Snapshot traces for Λ_{SI})**.** *Let $\mathcal{P}_0$ be a type-correct program in Λ_{SI}. Further, let R be a sequence of reductions*

$$\mathcal{H}_0, \mathcal{P}_0 \stackrel{\alpha_1}{\Longrightarrow} \ldots \stackrel{\alpha_n}{\Longrightarrow} \mathcal{H}_n, \mathcal{P}_n.$$

Then there exists an equivalent sequence R' of the form

$$\mathcal{H}_0, \mathcal{P}_0 \stackrel{\alpha'_1}{\Longrightarrow} \ldots \stackrel{\alpha'_n}{\Longrightarrow} \mathcal{H}_n, \mathcal{P}'_n$$

such that $\bar{\alpha}(R')$ is snapshot admissible.

Proof of 5.2: We apply the algorithm for reordering traces into a snapshot admissible form to the traces of R. Because the algorithm only requires the permutation of independent effects, by Lemma 5.10 the result is an equivalent reduction sequence with an admissible trace. □

As serial traces allow easier reasoning about transactional executions when compared to serializable traces, snapshot admissible traces are a simpler canonical form for snapshot traces. Reducing the number of possible interleavings for effects aids programmers in reasoning about the interaction of concurrently running transactions.

6 Formalization of Twilight

This section gives a formalization of the Twilight STM. Twilight STM splits the code of a transaction into a (functional) atomic phase, which behaves as in Λ_{STM}, and an (imperative) twilight phase. Code in the twilight phase executes before the decision about a transaction's fate (restart or commit) is made and can affect its outcome based on the actual state of the execution environment. The Twilight API has operations to detect and repair read inconsistencies as well as operations to overwrite previously written variables. It also permits the embedding of I/O operations which can be constructed in such a way that the I/O operation is executed exactly once.

In the actual implementation, twilight code may run concurrently with other transactions including their twilight code. Related work on formalizations of transactional memory([1], [6]) require transactions to be executed in a sequential fashion in order to abstract over the method to provide mutual exclusion to heap locations. The formalization we present here just restricts the possible interleaving of threads in such a way that the twilight code of each transaction runs solo, i.e. all other threads are stalled while a transaction executes its twilight code. It is therefore possible for a transaction to observe updates by other transactions when reaching the twilight zone.

6.1 Syntax

Figure 10 shows the syntax of Λ_{TWI}. In addition to the standard operations that were described in Section 3, there is now a special bind operator $\ggg$ for entering the twilight zone. The `error` value indicates that a thread is stuck in an erroneous state.

Figure 10 Syntax of Λ_{TWI}. Expressions marked in gray arise only during evaluation.

$$
\begin{array}{lll}
x \in \mathrm{Var} & & l \in \mathrm{Ref} \\
v \in \mathrm{Val} & ::= & l \mid \texttt{tt} \mid \texttt{ff} \mid () \mid \lambda x.e \mid \texttt{return}\ e \mid \texttt{error} \\
e \in \mathrm{Exp} & ::= & v \mid x \mid e\ e \mid \texttt{if}\ e\ e\ e \\
& & \mid \texttt{spawn}\ e \mid \texttt{atomic}\ e \mid e \gg\!= e \mid e \gg\!\gg e \\
& & \mid (e, W_i, R_i, i, e, \mathcal{H}) \mid (e, W_i, R_i, i, e, f) \\
& & \mid \texttt{newref}\ e \mid \texttt{readref}\ e \mid \texttt{writeref}\ e\ e \\
& & \mid \texttt{update}\ e\ e \mid \texttt{reread}\ e \mid \texttt{inconsistent}\ e \\
& & \mid \texttt{reload} \mid \texttt{ignoreUpdates} \mid \texttt{IOtoSTM}\ e \mid \texttt{retry}
\end{array}
$$

The syntax provides also repair operations for modifying the heap in the twilight zone. Variables that have been read or modified in the body of the transaction can be modified via `update` , and `reread` yields the value that a variable is currently associated with in the read set. The operation `inconsistent` compares the state of the transaction in the read set with its counterpart in the global heap. A consistent snapshot of the read set with the values that are currently in the heap can be obtained with `reload`. The `ignoreUpdates` operator allows a transaction to disregard updates by other transactions during conflict detection. By using `IOtoSTM`, an irrevocable expression can be embedded into the twilight zone. Finally, there is the `retry` method issues a restart of the transaction.

As in the type system for Λ_{STM}, Σ tracks the type of memory locations, and Γ tracks the type of variables. Figure 11 shows only the rules that differ from the ones in Figure 2.

The type system comprises now two other kinds of monad, the `TWI` and the `TXN` monad. The expressions that are evaluated as transactions are now all of type `TXN` monad as shown in rule T-Atomic. An instance of the `TXN` monad consists of both a transactional body and twilight code. The rule T-TwiBind deals with the switch form a transaction's body to the associated twilight zone. Expressions of type `TWI` τ may only be used within the twilight code of a transaction as they require special concurrency guarantees.

6.2 Operational Semantics

Figure 12 introduces further definitions for the operational semantics.

A transaction T_i is a tuple $(e, W_i, R_i, i, e, \mathcal{H})$. As before, it consists of the expression that is currently evaluated, the write set and the read set of the transaction, a (unique) transaction identifier, a copy of the whole expression that is to be evaluated transactionally for rollbacks, a copy of the heap taken at the beginning of the transaction or during a reload. Now, additionally, the Λ_{TWI} calculus requires another kind of transaction tuple which does not contain a heap copy, but a flag denoting the transaction's status. An *ok* flag indicates that a transaction's read set variables are consistent with the current heap, a *bad* flag denotes some inconsistency between the read set and the current heap.

An execution state consists of a heap, a thread pool with expressions that are concur-

Figure 11 Extension of type rules of Λ_{TWI}.

$$\text{Types:} \qquad \begin{array}{rcl} \tau & ::= & \texttt{bool} \mid () \mid \texttt{R}\,\tau \mid \tau \to \tau \mid \mu\,\tau \\ \mu & ::= & \texttt{IO} \mid \texttt{TXN} \mid \texttt{STM} \mid \texttt{TWI} \end{array}$$

$$\frac{}{\Sigma|\Gamma \vdash \texttt{error} : \tau}\ \text{T-Error} \qquad \frac{\Sigma|\Gamma \vdash e : \texttt{TXN}\,\tau}{\Sigma|\Gamma \vdash \texttt{atomic}\,e : \texttt{IO}\,\tau}\ \text{T-Atomic}$$

$$\frac{\Sigma|\Gamma \vdash e_1 : \texttt{STM}\,\tau \qquad \Sigma|\Gamma \vdash e_2 : \texttt{bool} \to \tau \to \texttt{TWI}\,\tau'}{\Sigma|\Gamma \vdash e_1 \ggg e_2 : \texttt{TXN}\,\tau'}\ \text{T-TwiBind}$$

$$\frac{\Sigma|\Gamma \vdash e : \texttt{TXN}\,\tau \qquad \Sigma|\Gamma \vdash e' : \texttt{TXN}\,\tau \qquad \Sigma \vdash W_i \qquad \Sigma \vdash R_i \qquad \Sigma \vdash \mathcal{H}}{\Sigma|\Gamma \vdash (e, W_i, R_i, i, e', \mathcal{H}) : \texttt{IO}\,\tau}\ \text{T-Txn}$$

$$\frac{\Sigma|\Gamma \vdash e : \texttt{TWI}\,\tau \qquad \Sigma|\Gamma \vdash e' : \texttt{TXN}\,\tau \qquad \Sigma \vdash W_i \qquad \Sigma \vdash R_i}{\Sigma|\Gamma \vdash (e, W_i, R_i, i, e', f) : \texttt{IO}\,\tau}\ \text{T-TwiTxn}$$

$$\frac{\Sigma|\Gamma \vdash e_1 : \texttt{R}\,\tau \qquad \Sigma|\Gamma \vdash e_2 : \tau}{\Sigma|\Gamma \vdash \texttt{update}\,e_1\,e_2 : \texttt{TWI}\,()}\ \text{T-Update} \qquad \frac{\Sigma|\Gamma \vdash e : \texttt{R}\,\tau}{\Sigma|\Gamma \vdash \texttt{reread}\,e : \texttt{TWI}\,\tau}\ \text{T-Reread}$$

$$\frac{\Sigma|\Gamma \vdash e : \texttt{R}\,\tau}{\Sigma|\Gamma \vdash \texttt{inconsistent}\,e : \texttt{TWI}\,\texttt{bool}}\ \text{T-Incons} \qquad \frac{}{\Sigma|\Gamma \vdash \texttt{reload} : \texttt{TWI}\,()}\ \text{T-Reload}$$

$$\frac{}{\Sigma|\Gamma \vdash \texttt{ignoreUpdates} : \texttt{TWI}\,()}\ \text{T-IgnoreUpdates} \qquad \frac{}{\Sigma|\Gamma \vdash \texttt{retry} : \texttt{TWI}\,\tau}\ \text{T-Retry}$$

$$\frac{\Sigma|\Gamma \vdash e : \texttt{IO}\,\tau}{\Sigma|\Gamma \vdash \texttt{IOtoSTM}\,e : \texttt{TWI}\,\tau}\ \text{T-Safe}$$

Figure 12 State extended with flags.

$$\begin{array}{lll} s & \in \text{State} & = \text{Heap} \times \text{Program} \times \text{ThreadId} \\ T & \in \text{Txn} & = \text{Exp} \times \text{Store} \times \text{Store} \times \text{Id} \times \text{Exp} \times \text{Store} \\ T' & \in \text{TwiTxn} & = \text{Exp} \times \text{Store} \times \text{Store} \times \text{Id} \times \text{Exp} \times \text{Flag} \\ f & \in \text{Flag} & = \{ok, bad\} \\ \alpha_i & \in \text{TxnEffect} & = \cdots \cup \{\alpha_i^t \mid \alpha \in \text{Effect}\} \end{array}$$

Figure 13 Semantics: Evaluation contexts and standard reduction rules.

Evaluation contexts:

$$\begin{array}{lcl} \mathcal{E} & ::= & []\ e \mid \texttt{if}\ []\ e\ e' \\ \mathcal{M} & ::= & \texttt{newref}\ e \mid \texttt{readref}\ [] \mid \texttt{writeref}\ []\ e \\ & \mid & \texttt{reread}\ [] \mid \texttt{update}\ []\ e \mid \texttt{inconsistent}\ [][] \gg\!= e \mid [] \gg\!\!\gg e \end{array}$$

Expression evaluation $\rightarrow$:

$$\begin{array}{rcl} (\lambda x.e)\ e' & \rightarrow & e[e'/x] \\ \texttt{if tt}\ e\ e' & \rightarrow & e \\ \texttt{if ff}\ e\ e' & \rightarrow & e' \end{array}$$

$$\frac{e \rightarrow e'}{\mathcal{E}[e] \rightarrow \mathcal{E}[e']}$$

Monadic evaluation $\curvearrowright$

$$\begin{array}{lcl} \texttt{return}\ e \gg\!= e' & \curvearrowright & e'\ e \\ \texttt{error} \gg\!= e & \curvearrowright & \texttt{error} \end{array}$$

$$\frac{e \rightarrow e'}{e \curvearrowright e'} \qquad \frac{m \curvearrowright m'}{\mathcal{M}[m] \curvearrowright \mathcal{M}[m']}$$

rently evaluated, and a thread identifier to denote the thread that is currently executing the twilight code of a transaction. The evaluation of a program starts in an initial configuration $\langle\rangle, \{0 \mapsto e\}, \cdot$ with an empty heap, a main thread t_0, and no twilight identifier set. A final configuration has the form $\mathcal{H}, \{0 \mapsto v_0, \ldots, t_n \mapsto v_n\}, -$. In contrast to the operational semantics of Λ_{STM}, an evaluation step in Λ_{TWI} can produce more than one effect (e.g. ReloadBad). The rules in Figures 13-19 define the semantics of the language constructs.

In Figure 13, $\mathcal{E}[\bullet]$ and $\mathcal{M}[\bullet]$ denote the evaluation context for expressions and monadic expressions, respectively. As an additional rule, the error statement is passed through the monad without further evaluation of statements.

The evaluation rules for the IO monad and the transaction body in Figure 14 are similar to the previous formalization. They only differ with respect to the twilight flag that is recorded in the system's state. Execution steps at top level or within a transaction are only permitted if no transaction is currently executing its twilight zone.

Figure 15 shows the evaluation of expressions within the twilight zone. Before committing, the transaction must switch from the `STM` monad to the `TWI` monad with the twilight bind $\gg\!\!\gg$. At this point, the heap is checked for updates to the references that are in the transaction's read set. There are two cases:

Rule TwiOk applies if the check is successful, this is, none of the heap locations read by the transaction have been updated by another transaction in the meantime. It

Figure 14 Operational semantics: IO and STM monad.

$$\frac{\mathcal{P}(t) = m \qquad m \curvearrowright m'}{\mathcal{H}, \mathcal{P}, - \xrightarrow{\epsilon^t} \mathcal{H}, \mathcal{P}\{t \mapsto m'\}, -} \text{ IO-Monad}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[\texttt{spawn}\ m] \qquad t' \text{ fresh}}{\mathcal{H}, \mathcal{P}, - \xrightarrow{sp^t(t')} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[\texttt{return}\ ()], t' \mapsto m\}, -} \text{ Spawn}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[\texttt{atomic}\ m] \qquad T = (m, \langle\rangle, \langle\rangle, i, m, \mathcal{H}) \qquad i \text{ fresh}}{\mathcal{H}, \mathcal{P}, - \xrightarrow{at_i^t} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[T]\}, -} \text{ Atomic}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(m, W_i, R_i, i, m', \mathcal{H}')] \qquad m \curvearrowright m''}{\mathcal{H}, \mathcal{P}, - \xrightarrow{\epsilon_t^i} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(m'', W_i, R_i, i, m', \mathcal{H}')]\}, -} \text{ STM-Monad}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{newref}\ e], W_i, R_i, i, m', \mathcal{H}')] \qquad l \notin \mathcal{P}, \mathcal{H}}{\mathcal{H}, \mathcal{P}, - \xrightarrow{\epsilon_t^i} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\mathcal{M}'[\texttt{return}\ l], W_i[l \mapsto (e, i)], R_i, i, m', \mathcal{H}')]\}, -} \text{ Alloc}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\,\texttt{writeref}\ l\ e], W_i, R_i, i, m', \mathcal{H}')]}{\mathcal{H}, \mathcal{P}, - \xrightarrow{\epsilon_t^i} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\mathcal{M}'[\texttt{return}\ ()], W_i[l \mapsto (e, i)], R_i, i, m', \mathcal{H}')]\}, -} \text{ Write}$$

$$\frac{\begin{array}{c}\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{readref}\ l], W_i, R_i, i, m', \mathcal{H}')] \\ l \notin \text{dom}(W_i) \cup \text{dom}(R_i) \qquad \mathcal{H}(l) = \mathcal{H}'(l) = (e, j)\end{array}}{\mathcal{H}, \mathcal{P}, - \xrightarrow{r_i^{t_i}(l)} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\mathcal{M}'[\texttt{return}\ e], W_i, R_i[l \mapsto (e, j)], i, m', \mathcal{H}')]\}, -} \text{ Read}$$

$$\frac{\begin{array}{c}\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{readref}\ l], W_i, R_i, i, m', \mathcal{H}')] \\ l \notin \text{dom}(W_i) \qquad R_i(l) = (e, i)\end{array}}{\mathcal{H}, \mathcal{P}, - \xrightarrow{\epsilon_t^i} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\mathcal{M}'[\texttt{return}\ e], W_i, R_i, i, m', \mathcal{H}')]\}, -} \text{ ReadRSet}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{readref}\ l], W_i, R_i, i, m', \mathcal{H}')] \qquad W_i(l) = (e, i)}{\mathcal{H}, \mathcal{P}, - \xrightarrow{\epsilon_t^i} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\mathcal{M}'[\texttt{return}\ e], W_i, R_i, i, m', \mathcal{H}')]\}, -} \text{ ReadWSet}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(m, W_i, R_i, i, m', \mathcal{H}')]}{\mathcal{H}, \mathcal{P}, - \xrightarrow{ab_i^t} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[\texttt{atomic}\ m']\}, -} \text{ Rollback}$$

Figure 15 Operational semantics: Control flow in the twilight zone.

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{return}\ e \ggg\!= m], W_i, R_i, i, m', \mathcal{H}')] \qquad check(R_i, \mathcal{H}) = ok}{\mathcal{H}, \mathcal{P}, - \xrightarrow{\epsilon_t^i} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\mathcal{M}'[m\ \texttt{tt}\ e], W_i, R_i, i, m', ok)]\}, t} \textsc{TwiOk}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{return}\ e \ggg\!= m], W_i, R_i, i, m', \mathcal{H}')] \qquad check(R_i, \mathcal{H}) = bad}{\mathcal{H}, \mathcal{P}, - \xrightarrow{\epsilon_i^t} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\mathcal{M}'[m\ \texttt{ff}\ e], W_i, R_i, i, m', bad)]\}, t} \textsc{TwiBad}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{reload}], W_i, R_i, i, m', ok)]}{\mathcal{H}, \mathcal{P}, t \xrightarrow{\epsilon_t^i} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\mathcal{M}'[\texttt{return}\ ()], W_i, R_i, i, m', ok)]\}, t} \textsc{ReloadOk}$$

$$\frac{\begin{array}{c}\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{reload}], W_i, R_i, i, m', bad)] \\ j \text{ fresh} \qquad R_j = \{l \mapsto \mathcal{H}(l) | l \in \mathrm{dom}(R_i)\}\end{array}}{\mathcal{H}, \mathcal{P}, t \xrightarrow{ab_i^t, at_j^t, \overline{r_j^t(l)}} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\mathcal{M}'[\texttt{return}\ ()], W_j, R_j, j, m', ok)]\}, t} \textsc{ReloadBad}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{ignoreUpdates}], W_i, R_i, i, m', f)]}{\mathcal{H}, \mathcal{P}, t \xrightarrow{\epsilon_t^i} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\mathcal{M}'[\texttt{return}\ ()], W_i, R_i, i, m', ok)]\}, t} \textsc{IgnoreUpdates}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\texttt{retry}\ , W_i, R_i, i, m', f)]}{\mathcal{H}, \mathcal{P}, t \xrightarrow{ab_i^t} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[\texttt{atomic}\ m']\}, -} \textsc{Retry}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\texttt{return}\ e, W_i, R_i, i, m', ok)] \qquad \mathcal{H}'' = \mathcal{H}[W_i']}{\mathcal{H}, \mathcal{P}, t \xrightarrow{co_i^t(\overline{l})} \mathcal{H}'', \mathcal{P}[t \mapsto \mathcal{M}[\texttt{return}\ e]], -} \textsc{Commit}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\texttt{return}\ e, W_i, R_i, i, m', bad)]}{\mathcal{H}, \mathcal{P}, t \xrightarrow{ab_i^t} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[\texttt{atomic}\ m']\}, -} \textsc{CommitFail}$$

Figure 16 Helper relation.

$$\frac{\forall l \in \mathrm{dom}(R_i) : R_i(l) = \mathcal{H}(l)}{check(R_i, \mathcal{H}) = ok} \qquad \frac{\exists l \in \mathrm{dom}(R_i) : R_i(l) \neq \mathcal{H}(l)}{check(R_i, \mathcal{H}) = bad}$$

$$\frac{\forall l \in \mathrm{dom}(R_i) \cap \mathrm{dom}(W_i) : R_i(l) = \mathcal{H}(l)}{check(R_i, W_i, \mathcal{H}) = ok} \qquad \frac{\exists l \in \mathrm{dom}(R_i) \cap \mathrm{dom}(W_i) : R_i(l) \neq \mathcal{H}(l)}{check(R_i, W_id, \mathcal{H}) = bad}$$

sets the twilight flag to *ok*.

Rule TwiBad applies if the check fails. It sets the transaction's twilight flag to *bad*. In the `TWI` monad, the transaction's twilight state can be set to *ok* with a `reload` or `ignoreUpdates`. Thus, the transaction can repair or ignore its inconsistencies and still commit successfully.

The definition of the helper function for the check is in Figure 16. The boolean value that is passed to the next statement m in the `TWI` monad reveals the outcome of the consistency check to the transaction's execution context. Further, the thread identifier in the global state is set to the identifier of the thread which executes the transaction.

If there are no inconsistencies, the `reload` operation does not change the internal state of the transaction. Otherwise, entries in the read set are replaced by their counterparts in the global heap. This corresponds semantically to an abort of the transaction, and the start of a new transaction which adopts the reads set and write set, as well as the execution context of the aborted predecessor. The annotated effects reflect it by emitting the abort effect for the transaction, the begin effect for the new transaction, and a list of all read effects as listed in the read set. Also, the transaction's state is now found consistent with respect to the current heap and is flagged with *ok*.

In a similar way, `ignoreUpdates` also puts the transaction into a committable state by setting the state flag to *ok*. Because no new values are observed, nor global operations performed, the empty effect is emitted to the trace.

When the twilight zone has been reduced to a return statement, the rule for commit transfers its heap modification as registered in the write set to the global heap (Commit). A corresponding commit effect which contains the locations of the modified variables is emitted, and the thread identifier in the global state which indicated that a twilight zone is exectued is reset.

If the transaction has been found inconsistent with respect to the global heap when entering its twilight zone, and it has not obtained an update of the read variables via `reload` or explicitly ignored updates via `ignoreUpdates`, the commit fails (CommitFail). The transaction formally aborts and is restarted completely.

With `IOtoSTM`, an statement to be evaluated in the IO monad can be lifted into the twilight zone of a transaction. The statement, with the exception of `atomic` expressions (cf. IOtoSTMErr), is evaluated in a top level environment. The effects are transferred to the enclosing transaction. When a new thread is spawn (IOtoSTMSpawn), it is added to the system's thread pool for later execution. IOtoSTMEnd returns to the execution context of the enclosing transaction.

Figure 18 shows the rules for repair operations in the twilight zone. Evaluating `inconsistent` l yields the result of comparing the value for l in the read set with the one in the global heap. Similarly to Write, the `update` operation replaces the value in the write set (Update), while the `reread` operation returns the value for a reference in the read set (Reread).

Errors are induced by invalid read or write operations inside the twilight zone as depicted in Figure 19. A read (or write) operation is illegal in the twilight code if its

Figure 17 Operational semantics: Embedding of I/O operations.

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{IOtoSTM}\ e], W_i, R_i, i, m, f)] \qquad e \neq \texttt{atomic}\ m' \qquad \mathcal{H}, \{t \mapsto e\}, - \xRightarrow{\alpha^t} \mathcal{H}, \{t \mapsto e'\}, -}{\mathcal{H}, \mathcal{P}, t \xRightarrow{\alpha_i^t} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\mathcal{M}'[\texttt{IOtoSTM}\ e'], W_i, R_i, i, m, f)]\}, t}\ \textsc{IOtoSTM}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{IOtoSTM}\ e], W_i, R_i, i, m, f)] \qquad \mathcal{H}, \{t \mapsto e\}, - \xRightarrow{\alpha^t} \mathcal{H}, \{t \mapsto e'; t' \mapsto e''\}, -}{\mathcal{H}, \mathcal{P}, t \xRightarrow{\alpha_i^t} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\mathcal{M}'[\texttt{IOtoSTM}\ e'], W_i, R_i, i, m, f)]; t' \mapsto e''\}, t}\ \textsc{IOtoSTMSpawn}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{IOtoSTM return}\ e], W_i, R_i, i, m, \mathcal{H})f]}{\mathcal{H}, \mathcal{P}, t \xRightarrow{\epsilon_t^i} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\mathcal{M}'[\texttt{return}\ e], W_i, R_i, i, m, f)]\}, t}\ \textsc{IOtoSTMEnd}$$

Figure 18 Operational semantics: Repair operations.

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{inconsistent}\ l], W_i, R_i, i, m', f)] \qquad R_i(l) = \mathcal{H}(l)}{\mathcal{H}, \mathcal{P}, t \xRightarrow{\epsilon_t^i} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\mathcal{M}'[\texttt{return ff}], W_i, R_i, i, m', f)]\}, t}\ \textsc{InconsFalse}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{inconsistent}\ l], W_i, R_i, i, m', \mathcal{H})f] \qquad R_i(l) \neq \mathcal{H}(l)}{\mathcal{H}, \mathcal{P}, t \xRightarrow{\epsilon_t^i} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\mathcal{M}'[\texttt{return tt}], W_i, R_i, i, m', f)]\}, t}\ \textsc{InconsTrue}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{update}\ l\ e], W_i, R_i, i, m, f)] \qquad l \in \mathrm{dom}(W_i)}{\mathcal{H}, \mathcal{P}, t \xRightarrow{\epsilon_t^i} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\mathcal{M}'[\texttt{return}\ ()], W_i[l \mapsto (e, i)], R_i, i, m, f)]\}, t}\ \textsc{Update}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{reread}\ l], W_i, R_i, i, m, f)] \qquad R_i(l) = (e, j)}{\mathcal{H}, \mathcal{P}, t \xRightarrow{\epsilon_t^i} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\mathcal{M}'[\texttt{return}\ e], W_i, R_i, i, m, f)]\}, t}\ \textsc{Reread}$$

Figure 19 Operational semantics: Error.

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{inconsistent}\ l], W_i, R_i, i, m', f)] \qquad l \notin \mathrm{dom}(R_i)}{\mathcal{H}, \mathcal{P}, t \overset{ab_i^t}{\Longrightarrow} \mathcal{H}, \mathcal{P}[t \mapsto \mathcal{M}[\texttt{error}]], -} \textsc{InconsErr}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{update}\ l\ e], W_i, R_i, i, m, f)] \qquad l \notin \mathrm{dom}(W_i)}{\mathcal{H}, \mathcal{P}, t \overset{ab_i^t}{\Longrightarrow} \mathcal{H}, \mathcal{P}[t \mapsto \mathcal{M}[\texttt{error}]], -} \textsc{UpdateErr}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{reread}\ l], W_i, R_i, i, m, f)] \qquad l \notin \mathrm{dom}(R_i)}{\mathcal{H}, \mathcal{P}, t \overset{ab_i^t}{\Longrightarrow} \mathcal{H}, \mathcal{P}[t \mapsto \mathcal{M}[\texttt{error}]], -} \textsc{RereadErr}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{IOtoSTM}\ \texttt{atomic}\ m'], W_i, R_i, i, m, f)]}{\mathcal{H}, \mathcal{P}, t \overset{ab_i^t}{\Longrightarrow} \mathcal{H}, \mathcal{P}[t \mapsto \mathcal{M}[\texttt{error}]], -} \textsc{IOtoSTMErr}$$

location has not been read (or written) in the preceding `STM` phase of the transaction. Another source for errors is the nesting of transactions within `IOtoSTM`. As there is no obvious good semantics for open nested transactions [7], we follow here the semantics that is specified for Haskell's STM and dynamically reject the evaluation of `atomic` m.

Errors abort the enclosing transaction. When they are propagated to the top-level, they terminate the execution of the associated thread.

Theorem 6.1 (Type soundness)**.** *The type system in Figure 11 is sound with respect to the operational semantics of* Λ_{TWI}*.*

6.3 Semantics of Twilight Transactions

Twilight STM provides not only an enriched interface for programming transactions, it also allows weakening of isolation semantics of transactions. In database transactions, it is common to have several levels of isolation. For software transactions, weakening of the isolation level can have undesirable and unexpected effects.

To aid the programmer in employing relaxed isolation semantics, all twilight transactions adhere to the principle of consistency. Therefore, the operational semantics does not allow zombie transactions that are doomed to fail, or exhibit all kind of undesired behavior due to violated invariants in inconsistent memory snapshots.

Lemma 6.1 (Consistency)**.** *A twilight transaction always operates on a consistent memory snapshot.*

Proof of 6.1: The consistency of the transaction's memory snapshot can only be violated by reading variables that were updated on the global heap since the begin of

the transaction. We therefore have to consider all rules that operate on the global heap. They are easy to identify as they emit read and commit effects.
Case distinction on rules accessing the global heap.

- *Case* ATOMIC: When starting the transaction, a copy of the global heap is obtained. This operation is atomic and cannot be interleaved by modifications of the heap.
- *Case* READ: The rule READ checks upon each first access to a reference if it is consistent with the variables that are have been read so far. Therefore, each reference is compared to its counterpart in the copy of the heap that has been acquired when starting the execution of the transaction. Only if the current heap contains the same value as the heap copy (i.e., the value has not changed), the read operation is successfully performed.
- *Case* RELOADBAD: The reload of the read set is performed in an atomic operation that cannot be interleaved with any update operation. Both the local copy of the heap and the read set are updated with the current values of the references.
- *Case* COMMIT: All update operations that are issued by a transaction get published to the global heap when commit via COMMIT. As the commit is performed in one indivisible operation, the heap's consistency is not violated, and no inconsistent state can be observed by another transaction.

End case distinction on rules accessing the global heap. □

Starting from a program which employs standard atomic blocks, how does adding a twilight zone influence the program's semantics? Given the guarantee of consistent memory snapshots, the programmer can specify the desired isolation semantics for each program in Λ_{TWI} individually, and obtain stronger semantics for a program. In the next sections, we show how operations in the twilight zone can define the isolation level of opacity and snapshot isolation by transforming STM monads from Λ_{STM} in Λ_{TWI}.

6.4 Opacity in Λ_{TWI}

Implementing opacity in TwilightSTM is straightforward by transforming the code statically with $[\![\cdot]\!]_o$. The transformation extends the atomic blocks with an (empty) twilight zone which simply returns the result of evaluating the STM monad in the block. All other expressions are not changed.

$$
\begin{aligned}
[\![x]\!]_o &= x \\
[\![\texttt{tt}]\!]_o &= \texttt{tt} \\
[\![\texttt{ff}]\!]_o &= \texttt{ff} \\
[\![()]\!]_o &= () \\
[\![\lambda x.e]\!]_o &= \lambda x.[\![e]\!]_o \\
[\![e_1\ e_2]\!]_o &= [\![e_1]\!]_o\ [\![e_2]\!]_o \\
[\![\texttt{if}\ e_1\ e_2\ e_3]\!]_o &= \texttt{if}\ [\![e_1]\!]_o\ [\![e_2]\!]_o [\![e_3]\!]_o \\
[\![\texttt{return}\ e]\!]_o &= \texttt{return}\ [\![e]\!]_o \\
[\![e_1 \gg\!= e_2]\!]_o &= [\![e_1]\!]_o \gg\!= [\![e_2]\!]_o \\
[\![\texttt{spawn}\ e]\!]_o &= \texttt{spawn}\ [\![e]\!]_o \\
[\![\texttt{newref}\ e]\!]_o &= \texttt{newref}\ [\![e]\!]_o \\
[\![\texttt{readref}\ e]\!]_o &= \texttt{readref}\ [\![e]\!]_o \\
[\![\texttt{writeref}\ e_1\ e_2]\!]_o &= \texttt{writeref}\ [\![e_1]\!]_o\ [\![e_2]\!]_o \\
[\![\texttt{atomic}\ m]\!]_o &= \texttt{atomic}\ ([\![m]\!]_o \ggg \lambda x.\lambda y.\texttt{return}\ y)
\end{aligned}
$$

Theorem 6.2 (Opacity for Twilight transactions)**.** *The execution trace of a program m in Λ_{STM} is equivalent in effects to a trace of the transformed program $[\![m]\!]_o$ in Λ_{TWI}.*

Proof of 6.2: The proof is done by induction on evaluation steps.

The rules IO-Monad, Spawn, Atomic, STM-Monad, Alloc, Write, Read, ReadWSet, and ReadRSet in Λ_{STM} have an equal evaluation rule with the same name in Λ_{TWI} which is taken when evaluating expressions in the `IO` and `STM` monad. The rules in Λ_{TWI} merely extend the system and transaction state with twilight flags. In all rules that define evaluation outside the `TWI` monad are the flags not set.

The scheduling for threads in Λ_{STM} can be simulated in Λ_{TWI} as the transaction which executes a twilight zone is running solo. It cannot be interleaved with evaluation steps from other threads.

The only differences arise when performing the commit operation in Λ_{STM} and its equivalent in Λ_{TWI}, namely the twilight bind and the following evaluation of the twilight zone.

We now consider the state of the system in Λ_{STM} where a thread has evaluated an atomic block to the transaction tuple $(\texttt{return}\ e, W_i, R_i, i, m', \mathcal{H}')$, and the scheduling chose this thread for the next step.

Case distinction on the applicable rules.

- *Case* Commit: The rule requires that $check(R_i, \mathcal{H}) = ok$. It then yields in the execution trace the following step:

$$
\begin{aligned}
& \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{return}\ e, W_i, R_i, i, m', \mathcal{H}')]\} \\
\overset{co_i^t(\bar{l})}{\Longrightarrow}\ & \mathcal{H}'', \mathcal{P}\{t \mapsto \mathcal{M}[\texttt{return}\ e]\}
\end{aligned}
$$

where $\mathcal{H}'' = \mathcal{H}[W_i]$ and $\bar{l} = \mathrm{dom}(W_i)$.

- *Case* ROLLBACK: If the check failed (i.e. $check(R_i, \mathcal{H}) = bad$), or a non-deterministic choice requires the transaction to abort, the execution trace continues with a rollback:

$$\begin{array}{rl} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{return}\ e, W_i, R_i, i, m', \mathcal{H}')]\} \\ \xrightarrow{ab_i^t} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[\texttt{atomic}\ m']\} \end{array}$$

End case distinction on the applicable rules.

Now, consider the rules that are applicable in Λ_{TWI} when evaluating the corresponding expression

$$(\texttt{return}\ e \ggg \lambda x.\lambda y.\texttt{return}\ y, W_i, R_i, i, m', \mathcal{H}')$$

.

Case distinction on the applicable rules.

- *Case* TWIOK: The rule requires that $check(R_i, \mathcal{H}) = ok$. The evaluation then proceeds with these steps:

$$\begin{array}{rl} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{return}\ e \ggg \lambda x.\lambda y.\texttt{return}\ y, W_i, R_i, i, m', \mathcal{H}')]\}, - \\ \xrightarrow{\epsilon_t^i} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[((\lambda x.\lambda y.\texttt{return}\ y)\ \texttt{tt}\ e, W_i, R_i, i, m', ok)]\}, t \\ \xrightarrow{\epsilon_t^i} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[((\lambda y.\texttt{return}\ y)\ e, W_i, R_i, i, m', ok)]\}, t \\ \xrightarrow{\epsilon_t^i} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{return}\ e, W_i, R_i, i, m', ok)]\}, t \\ \xrightarrow{co_i^t(\bar{l})} & \mathcal{H}', \mathcal{P}\{t \mapsto \mathcal{M}[\texttt{return}\ e]\}, - \end{array}$$

  with $\mathcal{H}' = \mathcal{H}[W_i]$ and $\bar{l} = \mathrm{dom}(W_i)$

- *Case* TWIBAD: The rule requires that $check(R_i, \mathcal{H}) = bad$. Hence, at commit the transaction failed verification because the empty twilight zone does not perform any repair or ignoring of inconsistencies.

$$\begin{array}{rl} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{return}\ e \ggg \lambda x.\lambda y.\texttt{return}\ y, W_i, R_i, i, m', \mathcal{H}')]\}, - \\ \xrightarrow{\epsilon_t^i} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[((\lambda x.\lambda y.\texttt{return}\ y)\ \texttt{ff}\ e, W_i, R_i, i, m', bad)]\}, t \\ \xrightarrow{\epsilon_t^i} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[((\lambda y.\texttt{return}\ y)\ e, W_i, R_i, i, m', bad)]\}, t \\ \xrightarrow{\epsilon_t^i} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{return}\ e, W_i, R_i, i, m', bad)]\}, t \\ \xrightarrow{ab_i^t} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[\texttt{atomic}\ m']\}, - \end{array}$$

- *Case* ROLLBACK: As in Λ_{STM}, the transaction aborts and restarts.

$$\begin{array}{rl} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{return}\ e \ggg \lambda x.\lambda y.\texttt{return}\ y, W_i, R_i, i, m', \mathcal{H}')]\}, - \\ \xrightarrow{ab_i^t} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[\texttt{atomic}\ m']\}, - \end{array}$$

Figure 20 Snapshot operations.

Syntax:

$$e \in \text{Exp} ::= \cdots \mid \texttt{wsetCons}$$

Type rules:

$$\frac{}{\Sigma|\Gamma \vdash \texttt{wsetCons} : \texttt{TWI}\,()}\ \text{T-WSetCons}$$

Operational semantics:

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{wsetCons}], W_i, R_i, i, m', f)] \qquad check(R_i, W_i, \mathcal{H}) = ok}{\mathcal{H}, \mathcal{P}, t \xrightarrow{\epsilon_t^i} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\mathcal{M}'[\texttt{return tt}], W_i, R_i, i, m', f)]\}, t}\ \text{WSetCons}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{wsetCons}], W_i, R_i, i, m', f)] \qquad check(R_i, W_i, \mathcal{H}) = bad}{\mathcal{H}, \mathcal{P}, t \xrightarrow{\epsilon_t^i} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\mathcal{M}'[\texttt{return ff}], W_i, R_i, i, m', f)]\}, t}\ \text{WSetIncons}$$

End case distinction on the applicable rules.

Each execution trace in Λ_{STM} has an equivalent counterpart in Λ_{TWI}: if the transaction commits successfully in Λ_{STM}, there is an equivalent execution trace in Λ_{TWI} which commits and results in the same final global state. In the same way, an abort in Λ_{STM} can be simulated by an abort in Λ_{TWI}.

For each of these execution traces, the effect traces are equivalent in effects, as the effect traces in Λ_{TWI} contain only additional empty effects. □

6.5 Snapshot isolation in Λ_{TWI}

To implement snapshot isolation, transactions need to operate on a consistent memory snapshot. Further, as explained in Section 5, the entries in the write set must be checked for intermediate updates between the begin of the transaction and its commit.

By Lemma 6.1, the operational semantics of Λ_{TWI} exacts memory consistency. Therefore, the twilight code just needs to specify operations that obviate lost updates.

To simplify the transformation in the formal calculus, we enlarge the formal language Λ_{TWI} with a new primitive, `wsetCons`. The operation `wsetCons` is testing for inconsistencies in the write set. In an implementation of TwilightSTM, this operation can easily be provided as a primitive. Alternatively, all references to the variables that are modified can be dynamically tagged, and tested individually for inconsistencies with `inconsistent` .

We can define a transformation from Λ_{SI} to Λ_{TWI} which preserves the operational semantics by yielding traces that are equivalent in effects. As with $[\![\cdot]\!]_o$, only the atomic

blocks are transformed:

$$\begin{aligned}[\![\texttt{atomic}\ m]\!]_s \quad = \quad & \texttt{atomic}\ [\![m]\!]_s \ggg\!= \lambda x.\lambda y.\texttt{wsetCons} \gg\!= \\ & \lambda b.\texttt{if}\ b\ (\texttt{ignoreUpdates} \gg\!= \lambda z.\texttt{return}\ y)\ (\texttt{retry}\ ))\end{aligned}$$

All other expressions are transformed recursively, analogously to $[\![\cdot]\!]_o$.

Theorem 6.3 (Snapshot Isolation for Twilight transactions)**.** *The execution trace of a program m in Λ_{SI} is equivalent in effects to a trace of the transformed program $[\![m]\!]_s$ in Λ_{TWI}.*

Proof of 6.3: As in the proof of 6.2, all rules but the commit rule in Λ_{SI} have an equivalent counterpart in Λ_{TWI}.

We again consider the possible execution steps at commit time in both formalizations and show that they yield equivalent results.

Consider the state of the system in Λ_{SI} where a thread has evaluated an atomic block to the transaction tuple $(\texttt{return}\ e, W_i, R_i, i, m', \mathcal{H}')$, and the scheduling chose this thread for the next step.
Case distinction on the applicable rules.

- *Case* Commit: The rule requires that $check(R_i, W_i, \mathcal{H}) = ok$. It then yields in the execution trace the following step:

  $$\mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{return}\ e, W_i, R_i, i, m', \mathcal{H}')]\} \overset{co_i^t(\bar{l})}{\Longrightarrow} \mathcal{H}'', \mathcal{P}\{t \mapsto \mathcal{M}[\texttt{return}\ e]\}$$

  where $\mathcal{H}'' = \mathcal{H}[W_i]$ and $\bar{l} = \mathrm{dom}(W_i)$.

- *Case* Rollback: If the check failed (i.e. $check(R_i, W_i, \mathcal{H}) = bad$), or a non-deterministic choice requires the transaction to abort, the execution trace is continued with a rollback:

  $$\mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{return}\ e, W_i, R_i, i, m', \mathcal{H}')]\} \overset{ab_i^t}{\Longrightarrow} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[\texttt{atomic}\ m']\}$$

End case distinction on the applicable rules.

As before, we distinguish between the rules that are applicable in Λ_{TWI} when evaluating the corresponding transformed expression

$$\begin{aligned} m \quad = \quad & \texttt{return}\ e \ggg\!= \lambda x.\lambda y.\texttt{wsetCons} \gg\!= \\ & \lambda b.\texttt{if}\ b\ (\texttt{ignoreUpdates} \gg\!= \lambda z.\texttt{return}\ y)\ (\texttt{retry}\ ) \end{aligned}$$

Case distinction on the applicable rules.

- *Case* TwiOk: The rule requires that $check(R_i, \mathcal{H}) = ok$. The evaluation then proceeds with these steps:

$$\begin{array}{rl} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(m, W_i, R_i, i, m', \mathcal{H}')]\}, - \\ \xrightarrow{\epsilon_t^i} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[((\lambda x.\lambda y.\texttt{wsetCons} \gg\!= \ldots)\ \texttt{tt}\ e, W_i, R_i, i, m', ok)]\}, t \\ \xrightarrow{\epsilon_t^i} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[((\lambda y.\texttt{wsetCons} \gg\!= \ldots)\ e, W_i, R_i, i, m', ok)]\}, t \\ \xrightarrow{\epsilon_t^i} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{wsetCons} \gg\!= \lambda b. \ldots, W_i, R_i, i, m', ok)]\}, t \end{array}$$

  As $check(R_i, \mathcal{H})$ implies $check(R_i, W_i, \mathcal{H})$, the execution continues with the rule WSetCons.

$$\begin{array}{rl} \xrightarrow{\epsilon_t^i} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{return tt} \gg\!= \lambda b. \ldots, W_i, R_i, i, m', ok)]\}, t \\ \xrightarrow{\epsilon_t^i} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[((\lambda b.\texttt{if}\ b\ \ldots)\ \texttt{tt}, W_i, R_i, i, m', ok)]\}, t \\ \xrightarrow{\epsilon_t^i} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{if tt}\ \ldots\ \ldots, W_i, R_i, i, m', ok)]\}, t \\ \xrightarrow{\epsilon_t^i} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{ignoreUpdates} \gg\!= \ldots, W_i, R_i, i, m', ok)]\}, t \\ \xrightarrow{\epsilon_t^i} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{return}\ () \gg\!= \ldots, W_i, R_i, i, m', ok)]\}, t \\ \xrightarrow{\epsilon_t^i} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[((\lambda z.\texttt{return}\ e)\ (), W_i, R_i, i, m', ok)]\}, t \\ \xrightarrow{\epsilon_t^i} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{return}\ e, W_i, R_i, i, m', ok)]\}, t \\ \xrightarrow{co_i^t(\bar{l})} & \mathcal{H}', \mathcal{P}\{t \mapsto \mathcal{M}[\texttt{return}\ e]\}, - \end{array}$$

- *Case* TwiBad: The rule requires that $check(R_i, \mathcal{H}) = bad$.

$$\begin{array}{rl} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(m, W_i, R_i, i, m', \mathcal{H}')]\}, - \\ \xrightarrow{\epsilon_t^i} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[((\lambda x.\lambda y.\texttt{wsetCons} \gg\!= \ldots)\ \texttt{ff}\ e, W_i, R_i, i, m', bad)]\}, t \\ \xrightarrow{\epsilon_t^i} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[((\lambda y.\texttt{wsetCons} \gg\!= \ldots)\ e, W_i, R_i, i, m', bad)]\}, t \\ \xrightarrow{\epsilon_t^i} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{wsetCons} \gg\!= \lambda b. \ldots, W_i, R_i, i, m', bad)]\}, t \end{array}$$

  The two possible consistency states for the write set yield these cases:

  Case distinction on $check(R_i, W_i, \mathcal{H})$.

– *Case* $check(R_i, W_i, \mathcal{H}) = ok$: The rule WSetCons is the only applicable rule.

$$\begin{array}{rl}
 & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{wsetCons} \gg\!= \lambda b. \ldots, W_i, R_i, i, m', bad)]\}, t \\
\xrightarrow{\epsilon_t^i} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{return tt} \gg\!= \lambda b. \ldots, W_i, R_i, i, m', bad)]\}, t \\
\xrightarrow{\epsilon_t^i} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[((\lambda b.\texttt{if } b \ \ldots\ )\ \texttt{tt}, W_i, R_i, i, m', bad)]\}, t \\
\xrightarrow{\epsilon_t^i} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{if tt } \ldots\ \ldots, W_i, R_i, i, m', bad)]\}, t \\
\xrightarrow{\epsilon_t^i} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{ignoreUpdates} \gg\!= \ldots, W_i, R_i, i, m', bad)]\}, t \\
\xrightarrow{\epsilon_t^i} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{return } () \gg\!= \ldots, W_i, R_i, i, m', ok)]\}, t \\
\xrightarrow{\epsilon_t^i} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[((\lambda z.\texttt{return } e)\ (), W_i, R_i, i, m', ok)]\}, t \\
\xrightarrow{\epsilon_t^i} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{return } e, W_i, R_i, i, m', ok)]\}, t \\
\xrightarrow{co_i^t(\bar{l})} & \mathcal{H}', \mathcal{P}\{t \mapsto \mathcal{M}[\texttt{return } e]\}, -
\end{array}$$

– *Case* $check(R_i, W_i, \mathcal{H}) = bad$: The rule WSetIncons is the only applicable rule.

$$\begin{array}{rl}
 & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{wsetCons} \gg\!= \lambda b. \ldots, W_i, R_i, i, m', bad)]\}, t \\
\xrightarrow{\epsilon_t^i} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{return ff} \gg\!= \lambda b. \ldots, W_i, R_i, i, m', bad)]\}, t \\
\xrightarrow{\epsilon_t^i} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[((\lambda b.\texttt{if } b \ \ldots\ )\ \texttt{ff}, W_i, R_i, i, m', bad)]\}, t \\
\xrightarrow{\epsilon_t^i} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{if ff } \ldots\ \ldots, W_i, R_i, i, m', bad)]\}, t \\
\xrightarrow{\epsilon_t^i} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{retry}, W_i, R_i, i, m', bad)]\}, t \\
\xrightarrow{ab_i^t} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[\texttt{atomic } m']\}, -
\end{array}$$

End case distinction on $check(R_i, W_i, \mathcal{H})$.

- *Case* Rollback: As in Λ_{STM}, the transaction is aborted and restarted.

$$\begin{array}{rl}
 & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(m, W_i, R_i, i, m', \mathcal{H}')]\}, - \\
\xrightarrow{ab_i^t} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[\texttt{atomic } m']\}, -
\end{array}$$

End case distinction on the applicable rules.

Again, no interleavings are possible when executing the twilight zone, so the traces for the execution in Λ_{TWI} and Λ_{SI} have an equivalent counterpart in the other formal language which yields effect traces that are equal in effects. □

6.6 Irrevocability in Λ_{TWI}

A major restriction of many transactional systems is the lack of support for executing irrevocable actions whose effects cannot in general be rolled back. These are in general I/O operations, such as system calls, that are to be executed inside transactions. It is possible to embed such calls into transactions in Λ_{TWI}. The following lemma illustrates how this can be done.

Lemma 6.2 (Irrevocability)**.** *Let*

$$T = m_1 \gg\!= \lambda x.\texttt{ignoreUpdates} \gg\!= \lambda y.\texttt{IOtoSTM}\, m_2 \gg\!= \lambda z.m_3$$

be a well-typed transaction in Λ_{TWI}. Then m_2 is evaluated at most once in each run of the transaction unless m_3 is contains an explicit call to `retry` *.*

Proof of 6.2: A transaction is restarted when it either contains a call to `retry` , or it applies Rollback during execution of the transactional body, or when its evaluation of the twilight zone ends in applying the rule CommitFail.

In the first case, by the assumption made in the lemma, only m_1 may contain a `retry` operator. Similarly, in the second case, the rule Rollback can only be applied when evaluating m_1. In both cases, the monadic expression m_2 is not executed before the restart is performed.

Consider the following cases upon evaluating `ignoreUpdates`:
Case distinction on entering the twilight zone.

- *Case* TwiOk: The state flag is set to *ok*. Then, the state is unchanged in `ignoreUpdates`.
- *Case* TwiBad: The state flag is set to *bad*. In IgnoreUpdates, the flag is finally set to *ok*.

End case distinction on entering the twilight zone.

There is no evaluation rule which switches the transaction's flag to the *bad* state. Hence, if no error is thrown or no call to `retry` occurs in m_3, the rule Commit is applied, and the transaction commits successfully. Hence, the monadic expression m_2 is only evaluated once. □

Besides `ignoreUpdates`, also the `reload` operator allows a transaction to switch into an "irrevocable mode" while yielding an updated snapshot of the read set. These operations also interact in such a way that calling `ignoreUpdates` before `reload` prevents the transaction from obtaining the current, possibly updated values. A programmer should therefore take care that each execution path in the twilight zone has preferably only one of these operations. The implementation of Twilight STM in Haskell ensures this with its parametrized monads and a special *Safe* monad with irrevocable semantics.

6.7 The power of Twilight operations

In the last sections we have shown how different semantics can be implemented for a transaction by employing twilight operations such as `ignoreUpdates`, `reload`, or `wsetCons`.

The programmer should be aware that these operations can also be (mis-) used to implement semantics that are usually undesirable. For example, the transaction

$$\texttt{atomic}\,\{\texttt{readref}\,x \gg\!= \lambda v.\,\texttt{writeref}\,x\,(v+1) \gg\!\gg\!= \lambda z.\texttt{ignoreUpdates}\}$$

can cause lost updates if another transaction commits to x concurrently.

Though, to simplify the reasoning about the interleaving and possible interaction of transactions, Λ_{TWI} provides only consistent memory snapshots for each transaction. This means that neither read skews nor non-repeatable reads can be observed. The interleaving can partially be observed by inspecting the read set's state and updated values, and the twilight zone may then react on these observations as the programmer specified.

References

[1] Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of transactional memory and automatic mutual exclusion. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 63–74, San Francisco, California, USA, 2008. ACM.

[2] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 1–10, San Jose, California, United States, 1995. ACM.

[3] David Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *Proceedings of the 20th International Symposium on Distributed Computing, DISC 2006*, LNCS 4167, pages 194–208. Springer, 2006.

[4] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In Siddhartha Chatterjee and Michael L. Scott, editors, *PPOPP*, pages 175–184. ACM, 2008.

[5] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *Sixteenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–60, Chicago, IL, USA, June 2005. ACM Press.

[6] Katherine F. Moore and Dan Grossman. High-level small-step operational semantics for transactions. In Phil Wadler, editor, *Proc. 35th ACM Symp. POPL*, pages 51–62, San Francisco, California, USA, January 2008. ACM.

[7] J. Eliot B. Moss. Open nested transactions: Semantics and support. Poster presented at Workshop on Memory Performance Issues (WMPI 2006), Austin, TX, February 2006.

[8] Gerhard Weikum and Gottfried Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[9] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.