Appendix

Table of Contents

All the cross references are hyper-linked.

$\mathcal{T}$:				
	$lock$: $Lock$			
	$clock$: $SCounter$			
	ver: $BasicRegister$			
$\mathcal{P}$:				
$L_1 \triangleright$	$lock.lock_{T_1}()$		$L_2 \triangleright$	$lock.lock_{T_2}()$
$C_1 \triangleright$	$v_1 = clock.iaf_{T_1}()$	$\|\|$	$C_2 \triangleright$	$v_2 = clock.iaf_{T_2}()$
$R_1 \triangleright$	$ver.write_{T_1}(v_1)$		$R_2 \triangleright$	$ver.write_{T_2}(v_2)$
$U_1 \triangleright$	$lock.unlock_{T_1}()$		$U_2 \triangleright$	$lock.unlock_{T_2}()$

Figure 12. Example Specification π

8 Simple Example

We introduce the program logic via a simple example. In this section, we present, first, an example specification in a subset of the specification language, then, the simplified program logic and finally, the deduction of a lemma for the example specification.

8.1 Algorithm Specification

Figure 12 specifies a simple algorithm that updates a register to ascending version numbers. In fact, it is a miniature version of the TL2 commit procedure. This specification has two sections: the type declaration section at the top and the concurrent program section at the bottom. In general, a specification can have a procedure definition section and call procedures that we postpone to the next section.

The type declaration section declares the *type* of each synchronization object used by the concurrent program. Three object types are used in this program: lock *Lock*, strong counter *SCounter* and basic register *BasicRegister*. Lock and strong counter are linearizable object types and basic register is a basic object type. In the general sense, linearizable objects can maintain *consistency* even if they are accessed *concurrently* while basic objects maintain consistency if they are not accessed concurrently. A register has two methods: $write$ and $read$. For example, $r.write(v)$ writes the value v to r, while $x = r.read()$ reads the value of r and binds x to that value. The language enforces unique binding for variables. A lock has two methods *lock* and *unlock* that lock and unlock it respectively. A strong counter has two methods: $read$ and iaf (increment-and-fetch). For a strong counter c, $x = c.read()$ reads the value of c and binds x to that value and $x = c.iaf()$ increments and then reads the value of c and binds x to that value. The objects $lock$, $clock$ and ver are declared of *Lock*, *SCounter*, and *BasicRegister* types.

The second section is the concurrent program. It is the parallel composition of a set of sequential programs. In this specification, there are two sequential programs where every statement is a method call. A method call is of the form $l \triangleright\ x = o.n_\tau(u)$ where l is the unique label of the method call. We define the following functions on labels that are immediately derived from the specification. obj_π maps l to the receiving object o, $name_\pi$ maps l to the method name n, $thread_\pi$ maps l to the calling thread identifier τ, $arg1_\pi$ maps l to the first argument u (that is either a variable x or a value v), and $retv_\pi$ maps l to the return variable x. The function $cond_\pi$ maps l to the *enclosing condition* of the method call labeled l. In this specification, we do not have if-then-else statements, therefore, $cond_\pi(l) = true$ for every label l. Every specification π, defines a *program order* $\rightarrow_\pi$ on the labels. Intuitively, $l_1 \rightarrow_\pi l_2$ means that the specification requires that if both l_1 and l_2 are executed, then l_1 must be executed before l_2. In this specification, we assume sequential consistency. Therefore, the program order $\rightarrow_\pi$ simply represents the order of labels in the program. We postpone relaxed order of method calls to next later section.

CONTROL

$$\frac{}{\pi, \Gamma \vdash exec(l) \Leftrightarrow cond_\pi(l)}$$

ID

$$\frac{obj_\pi(l) = o \qquad name_\pi(l) = n \qquad thread_\pi(l) = \tau \qquad arg1_\pi(l) = u \qquad retv_\pi(l) = x \qquad \pi, \Gamma \vdash exec(l)}{\pi, \Gamma \vdash obj(l) = o \ \wedge\ name(l) = n \ \wedge\ thread(l) = \tau \ \wedge\ arg1(l) = u \ \wedge\ retv(l) = x}$$

P2X

$$\frac{l_1 \rightarrow_\pi l_2 \qquad \pi, \Gamma \vdash exec(l_1) \qquad \pi, \Gamma \vdash exec(l_2)}{\pi, \Gamma \vdash l_1 \prec l_2}$$

SRC

$$\frac{\pi, \Gamma \vdash exec(l) \qquad \pi, \Gamma \vdash obj(l) = o \qquad \pi, \Gamma \vdash name(l) = n \qquad Calls_\pi(o, n) = \{\overline{l_i}\}}{\pi, \Gamma \vdash \bigvee_{i=1..n} l = l_i}$$

Each rule has the side condition $\pi = (\mathcal{T}, \mathcal{D}, \mathcal{P})$

Figure 13. Structure Inference Rules.

X2L

$$\frac{\mathcal{T}(o) \in LT \qquad \pi, \Gamma \vdash obj(l) = obj(l') = o \qquad \pi, \Gamma \vdash l \prec l'}{\pi, \Gamma \vdash l \prec_o l'}$$

XLTRANS

$$\frac{\pi, \Gamma \vdash l_1 \prec l_2 \qquad \pi, \Gamma \vdash l_2 \prec_o l_3 \qquad \pi, \Gamma \vdash l_3 \prec l_4}{\pi, \Gamma \vdash l_1 \prec l_4}$$

Each rule has the side condition $\pi = (\mathcal{T}, \mathcal{D}, \mathcal{P})$

Figure 14. Basic inference rules.

CountSeq

$$\frac{\begin{array}{c}\mathcal{T}(o) = SCounter \\ \pi,\Gamma \vdash exec(l_1) \ \wedge \ obj(l_1) = o \ \wedge \ name(l_1) = iaf \\ \pi,\Gamma \vdash exec(l_2) \ \wedge \ obj(l_2) = o \\ \pi,\Gamma \vdash retv(l_1) < retv(l_2)\end{array}}{\pi,\Gamma \vdash l_1 \prec_o l_2}$$

LockUnlockPair

$$\frac{\begin{array}{c}\mathcal{T}(o) = Lock \\ \pi,\Gamma \vdash isOwnerRespect(o) \\ \pi,\Gamma \vdash isLock_o(l_{l_1}) \qquad \pi,\Gamma \vdash isUnlock_o(l_{u_2}) \\ \pi,\Gamma \vdash l_{l_1} \prec_o l_{u_2}\end{array}}{\begin{array}{l}\pi,\Gamma \vdash \exists \ell_{u_1}, \ell_{l_2}: \\ isUnlock_{lo}(\ell_{u_1}) \ \wedge \ thread(\ell_{u_1}) = thread(l_{l_1}) \ \wedge \\ isLock_{lo}(\ell_{l_2}) \ \wedge \ thread(\ell_{l_2}) = thread(l_{u_2}) \ \wedge \\ \ell_{u_1} \prec_o \ell_{l_2}\end{array}}$$

$$isLock_o(l) \Leftrightarrow \quad exec(l) \ \wedge \ obj(l) = o \ \wedge \ name(l) = lock$$

$$isUnlock_o(l) \Leftrightarrow \quad exec(l) \ \wedge \ obj(l) = o \ \wedge \ name(l) = unlock$$

$$\begin{array}{l}isOwnerRespect(o) \Leftrightarrow \\ \quad \forall \ell: isUnlock_o(\ell) \Rightarrow \exists \ell': \\ \qquad isLock_o(\ell') \wedge \\ \qquad thread(\ell') = thread(\ell) \ \wedge \ \ell' \prec \ell \ \wedge \\ \qquad \forall \ell'': \\ \qquad\quad (isUnLock_o(\ell'') \wedge \\ \qquad\quad thread(\ell'') = thread(\ell)) \\ \qquad \Rightarrow \\ \qquad\quad \ell'' \prec \ell' \ \vee \ \ell \preceq \ell''\end{array}$$

Each rule has the side condition $\pi = (\mathcal{T}, \mathcal{D}, \mathcal{P})$

Figure 15. Synchronization Object Inference Rules.

8.2 Program Logic

Consider the two method calls labeled R_1 and R_2 in the specification (Figure 12). We will prove the following theorem that states that if the version that R_1 writes is less than the version that R_2 writes, then R_1 is executed before R_2. Although the statement of the lemma is simple, similar to the TM correctness assertions, it involves execution order and its proof involves linearization order of synchronization objects.

Lemma 8.1. $\pi, \cdot \vdash (arg1(R_1) < arg1(R_2)) \Rightarrow (R_1 \prec R_2)$.

Let us have an informal proof of the lemma first. We use the following five rules. First, the *program-order-preservation* property states that the program order is preserved in the execution order. Second, the *real-time-preservation* property states that the execution order is preserved in the linearization order. Third, the *execution-linearization-transitivity* property states that if l_1 is executed before l_2, l_2 is linearized before l_3 and l_3 is executed before l_4, then l_1 is executed before l_4. Forth, the *lock-unlock-pair* property states that if ownership of a lock l is respected and a *lock* method call on l (by a thread T_1) is linearized before an *unlock* method call on l (by a thread T_2), then an *unlock* method call on l by T_1 is linearized before a *lock* method call on l by T_2. Intuitively, ownership for a lock l is respected, if and only if every thread unlocks l only if it has already locked l and has not unlocked l since it has locked l. This specification π trivially respects ownership for its *lock* object. Fifth, the *count-sequence* property states that for a strong counter o, if the return value of an *iaf* method call on o is less than the return value of another method call on o, then the former is linearized before the latter.

We assume that (1) The argument of R_1 is less than the argument of R_2 and show that R_1 is executed before R_2. From the specification π, we have that (2) The argument of R_1 is the return value of C_1 and (3) the argument of R_2 is the return value of C_2. Thus, from [1], [2] and [3], we have that (4) the return value of C_1 is less than the return value of C_2. From π, we have that (5) C_1 and C_2 are *iaf* method calls on *clock* that is a strong counter. Thus, by count-sequence property on [5] and [4], we have that (6) C_1 is linearized before C_2. From π, we have (7) L_1 is before C_1 in the program and (8) C_1 is before U_2 in the program. By program-order-preservation on [7] and [8], we have that (9) L_1 is executed before C_1 and (10) C_2 is executed before U_2. By execution-linearization-transitivity property on [9], [6] and [10], we can conclude that (11) L_1 is executed before U_2. From π, we have (12) L_1 and U_2 are respectively *lock* and *unlock* method calls by threads T_1 and T_2 on the object *lock* that is of the linearizable type *Lock*. By the real-time-preservation property on [11], we have that (13) L_1 is linearized before U_2. By the lock-unlock-pair property on [12] and [13], we have that (14) an *unlock* method call by T_1 is linearized before a *lock* method call by T_2. From π, we have that (15) The *unlock* method call by T_1 is U_1 and (16) The *lock* method call by T_2 is L_2. Thus, from [14], [15] and [16], we have that (17) U_1 is linearized before L_2. From π, we have (18) R_1 is before U_1 in the program and (19) L_1 is before R_2 in the program. From the program-order-preservation property on [18] and [19], we have that (20) R_1 is executed before U_1 and (21) L_2 is executed before R_2. By the transitivity property on [20], [17] and [21], we have that R_1 is executed before R_2.

Now, let us introduce our logic and formalize the proof. The judgements of the logic are of the form $\pi, \Gamma \vdash \mathcal{A}$, where π is a specification, Γ is a list of assertions and $\mathcal{A}$ is an assertion. We use $\cdot$ to denote the empty list of assertions. Intuitively, a judgement $\pi, \Gamma \vdash \mathcal{A}$ states that in the context of the assertions Γ, the specification π has the property $\mathcal{A}$. The assertions are first-order logic assertions that involve the unary predicate *exec*, the binary predicates $\prec$ (*execution order*) and $\prec_o$ (*linearization order* of linearizable object o) and functions *obj*, *name*, *thread*, *arg*1 and *retv*. The assertion $exec(l)$ states that the method call labeled l is executed. The assertion $l_1 \prec l_2$ states that l_1 is executed before l_2. Any

concurrent execution on a linearizable object is equivalent to a correct sequential execution. The total order of method calls in the equivalent sequential execution is called the linearization order. For every linearizable object o, the assertion $l_1 \prec_o l_2$ states that l_1 is before l_2 in the linearization order of o. As π declares *lock* and *clock* as instances of linearizable types, the linearization orders of *lock* and *clock* are denoted by $\prec_{lock}$ and $\prec_{clock}$. We also use the equivalence relation on expressions and labels. The functions $obj(l)$, $name(l)$, $thread(l)$, $arg1(l)$, and $retv(l)$ map a label l to the receiving object, method name, calling thread identifier, the first argument and the return value of the method call labeled l.

Lemma 8.1 expresses a property of every execution of π, yet the soundness of the logic makes us able to prove it by reasoning about π alone. We consider an arbitrary execution of the specification. Given some facts about an execution, the inference rules let us derive more facts about that execution. The logic has four sets of inference rules: classical first-order logic inference rules, structure inference rules that axiomatize the association of the specification and the assertions, basic inference rules that axiomatize the properties of the execution and linearization orders and their interdependence and synchronization object inference rules that axiomatize the properties of common synchronization object types. We showcase a subset of structure inference rules in Figure 13, a subset of basic inference rules in Figure 14, and a subset of synchronization object inference rules in Figure 15.

The rule Control states that a method call is executed if and only if its enclosing condition is satisfied. The introduction rule Id states that the components (object, name, etc.) of a method call in the execution originate from the components of the method call in the program. The rule P2X states the program-order-preservation property. If a method call l_1 is ordered before a method call l_2 in the program, and methods l_1 and l_2 are executed, then l_1 is executed before l_2. The rule Src intuitively states that every executed method originates from a call site in the specification. Let $Calls_\pi(o,n)$ denote the set of labels of call sites where method name n is called on the object name o in the specification π. If the object and the name of an executed method call labeled l are o and n respectively, then l is equal to one of the labels in $Calls_\pi(o,n)$. For presentation purposes, this small example does not involve procedure calls and hence the rules Control, Id, and Src are simplified.

The rule X2L states the real-time-preservation property. The execution order of two method calls on a linearizable object is preserved in the linearization order. *LT* denotes the set of linearizable object types. The rule XLTrans states the execution-linearization-transitivity property defined above. Similarly, the rule LockUnlockPair and the rule CountSeq state the lock-unlock-pair and count-sequence properties defined above. The rule LockUnlockPair is derived from the fact that if the ownership of a lock is respected, its linearization order is a sequence of pairs of *lock* and *unlock* method calls by the same thread. The rule CountSeq is derived from the fact that the return value of method calls in the linearization order of a strong counter is non-decreasing.

8.3 Deduction

Now, let us see how the above informal reasoning can be formalized using inference rules. Let

$$\Gamma = arg1(R_1) < arg1(R_2) \tag{18}$$

Based on the classical condition introduction rule, to prove Lemma 8.1, we need to show that

$$\pi, \Gamma \vdash R_1 \prec R_2 \tag{19}$$

From 18, we have

$$\pi, \Gamma \vdash arg1(R_1) < arg1(R_2) \tag{20}$$

As mentioned before, there is no if-then-else in this specification; therefore, the enclosing condition of every label is trivially *true*. Thus, by the rule Control, we have

$$\pi, \Gamma \vdash exec(L_1) \tag{21}$$
$$\pi, \Gamma \vdash exec(C_1) \tag{22}$$
$$\pi, \Gamma \vdash exec(R_1) \tag{23}$$
$$\pi, \Gamma \vdash exec(U_1) \tag{24}$$
$$\pi, \Gamma \vdash exec(L_2) \tag{25}$$
$$\pi, \Gamma \vdash exec(C_2) \tag{26}$$
$$\pi, \Gamma \vdash exec(R_2) \tag{27}$$
$$\pi, \Gamma \vdash exec(U_2) \tag{28}$$

From the rule Id on 23, 27, 22, 26, and the specification π, we have

$$\pi, \Gamma \vdash arg1(R_1) = v_1 \tag{29}$$
$$\pi, \Gamma \vdash arg1(R_2) = v_2 \tag{30}$$
$$\pi, \Gamma \vdash retv(C_1) = v_1 \tag{31}$$
$$\pi, \Gamma \vdash retv(C_2) = v_2 \tag{32}$$

From the symmetry and transitivity of equivalence on [29], [30], [31], [32], we have

$$\pi, \Gamma \vdash arg1(R_1) = retv(C_1) \tag{33}$$
$$\pi, \Gamma \vdash arg1(R_2) = retv(C_2) \tag{34}$$

By substitution of 33 and 34 on [20], we have

$$\pi, \Gamma \vdash retv(C_1) < retv(C_2) \tag{35}$$

By the rule Id on 22, and the specification π, we have

$$\pi, \Gamma \vdash obj(C_1) = clock \tag{36}$$
$$\pi, \Gamma \vdash name(C_1) = iaf \tag{37}$$

By the rule Id on 26, and the specification π, we have

$$\pi, \Gamma \vdash obj(C_2) = clock \tag{38}$$

From rule CountSeq on 22, 36, 37, 26, 38, 35, we have

$$\pi, \Gamma \vdash C_1 \prec_{clock} C_2 \tag{39}$$

that is C_1 is linearized before C_2. The next step is to use rule P2X. From π, we have

$$L_1 \rightarrow_\pi C_1 \tag{40}$$
$$C_2 \rightarrow_\pi U_2 \tag{41}$$

By the rule P2X on 40, 21 and 22, we have

$$\pi, \Gamma \vdash L_1 \prec C_1 \tag{42}$$

Similarly, by the rule P2X on 41, 26 and 28, we have

$$\pi, \Gamma \vdash C_2 \prec U_2 \tag{43}$$

By the rule XLTrans on 42, 39 and 43, we have

$$\pi, \Gamma \vdash L_1 \prec U_2 \tag{44}$$

By the rule Id on 21, and the specification π, we have

$$\pi, \Gamma \vdash obj(L_1) = lock \tag{45}$$
$$\pi, \Gamma \vdash name(L_1) = lock \tag{46}$$
$$\pi, \Gamma \vdash thread(L_1) = T_1 \tag{47}$$

Similarly, by the rule Id on 28, and the specification π, we have

$$\pi, \Gamma \vdash obj(U_2) = lock \tag{48}$$
$$\pi, \Gamma \vdash name(U_2) = unlock \tag{49}$$
$$\pi, \Gamma \vdash thread(U_2) = T_2 \tag{50}$$

From rule X2L on 44, 45 and 48, we have

$$\pi, \Gamma \vdash L_1 \prec_{lock} U_2 \tag{51}$$

Now, we use the rule LockUnlockPair. The proof of ownership respect can be done using the presented rules. For the sake of brevity, we skip the proof of ownership respect.

$$\pi, \Gamma \vdash isOwnerRespecting(lock) \tag{52}$$

From the definition of *isLock* on 21, 45 and 46, we have

$$\pi, \Gamma \vdash isLock_{lock}(L_1) \tag{53}$$

From the definition of *isUnlock* on 28, 48 and 49, we have

$$\pi, \Gamma \vdash isUnlock_{lock}(U_2) \tag{54}$$

By the rule LockUnlockPair on 52, 53, 54, 51, and then substitution with 47 and 50, we have

$$\pi, \Gamma \vdash \exists \ell_{u_1}, \ell_{l_2} : isUnlock_{lock}(\ell_{u_1}) \ \wedge \ thread(\ell_{u_1}) = T_1 \ \wedge \ isLock_{lock}(\ell_{l_2}) \ \wedge \ thread(\ell_{l_2}) = T_2 \ \wedge \ \ell_{u_1} \prec_{lock} \ell_{l_2} \tag{55}$$

After skolemization of ℓ_{u_1} and ℓ_{l_2} with l_{u_1} and l_{l_2}, we have

$$\pi, \Gamma \vdash isUnlock_{lock}(l_{u_1}) \tag{56}$$
$$\pi, \Gamma \vdash thread(l_{u_1}) = T_1 \tag{57}$$
$$\pi, \Gamma \vdash isLock_{lock}(l_{l_2}) \tag{58}$$
$$\pi, \Gamma \vdash thread(l_{l_2}) = T_2 \tag{59}$$
$$\pi, \Gamma \vdash l_{u_1} \prec_{lock} l_{l_2} \tag{60}$$

From the definition of *isUnlock* on 56, we have

$$\pi, \Gamma \vdash exec(l_{u_1}) \tag{61}$$
$$\pi, \Gamma \vdash obj(l_{u_1}) = lock \tag{62}$$
$$\pi, \Gamma \vdash name(l_{u_1}) = unlock \tag{63}$$

From π, we have

$$Calls_\pi(lock, unlock) = \{U_1, U_2\} \tag{64}$$

By the rule Src on 61, 62, 63, and 64, we have

$$\pi, \Gamma \vdash l_{u_1} = U_1 \ \vee \ l_{u_1} = U_2 \tag{65}$$

Using negation introduction, from 50 and 57, we have

$$\pi, \Gamma \vdash \neg(l_{u_1} = U_2) \tag{66}$$

By disjunction syllogism on 65 and 66, we have

$$\pi, \Gamma \vdash l_{u_1} = U_1 \tag{67}$$

Similarly, using the rule Src, we can show that

$$\pi, \Gamma \vdash l_{l_2} = L_2 \tag{68}$$

By substitution of 67 and 68 to 60, we have

$$\pi, \Gamma \vdash U_1 \prec_{lock} L_2 \tag{69}$$

From π, we have

$$R_1 \rightarrow_\pi U_1 \tag{70}$$
$$L_2 \rightarrow_\pi R_2 \tag{71}$$

By the rule P2X on 70, 23 and 24, we have

$$\pi, \Gamma \vdash R_1 \prec U_1 \tag{72}$$

By the rule P2X on 71, 25 and 27, we have

$$\pi, \Gamma \vdash L_2 \prec R_2 \tag{73}$$

By the rule XLTrans on 72, 69, and 73, we have

$$\pi, \Gamma \vdash R_1 \prec R_2 \tag{74}$$

9 Algorithm Description

In this section, we extend the algorithm description syntax presented in the main body of the paper.

Syntax Extension. We define **foreach** statement as a syntactic sugar. The **foreach** statement iterates over sets and maps.

Consider a bounded *set* of type *Set*. The following **foreach** statement executes the statement s for each member i of set.

$$c \triangleright \textbf{ foreach } (i \in set) \quad s \tag{75}$$

Let b be a fresh variable name. We define $sIter(s,i)$, the ith iteration, as follows:

$$\begin{array}{rcl} sIter(s,i) & = & c_i \triangleright\ b_i = set.contains(i), \\ & & \textbf{if } (b_i) \\ & & \quad sIndexed(s,i) \end{array} \tag{76}$$

where $sIndexed(s,i)$ denotes a transformation of s where every label c is replaced by c_i and every variable x that is assigned in s is replaced by x_i. The **foreach** statement is a syntactic sugar for

$$\begin{array}{l} sIter(s,0), \\ sIter(s,1), \\ sIter(s,2), \\ \ldots \\ sIter(s,max) \end{array} \tag{77}$$

where max is the maximum value stored in the set.

Similarly, consider a bounded *map* of type *Map*. The following **foreach** statement executes the statement s for each mapping i to v in map.

$$c \triangleright \textbf{ foreach } ((i,v) \in map) \quad s \tag{78}$$

We define $mIter(s,i)$, the $i$the iteration, as follows:

$$\begin{array}{rcl} mIter(s,i) & = & c_i \triangleright\ v_i = map.get(i), \\ & & \textbf{if } (v_i \neq \bot) \\ & & \quad mIndexed(s,i) \end{array} \tag{79}$$

where $mIndexed(s,i)$ denotes a transformation of s where every label c is replaced by c_i, v is replaced with v_i, and every variable x that is assigned in s is replaced by x_i. The **foreach** statement is a syntactic sugar for

$$\begin{array}{l} mIter(s,0), \\ mIter(s,1), \\ mIter(s,2), \\ \ldots \\ mIter(s,max) \end{array} \tag{80}$$

where max is the maximum key.

Transaction Syntax. A transactional memory description π_{TM} is a particular case of an algorithm description $(\mathcal{T}, \mathcal{D}_{TM}, \mathcal{P}_{TM})$ where

$$\begin{array}{rcl} \mathcal{D}_{TM} & = & \textbf{def } init_t()\ s_0, r_0, \\ & & \textbf{def } read_t(i)\ s_1, r_1, \\ & & \textbf{def } write_t(i,v)\ s_2, r_2, \\ & & \textbf{def } commit_t()\ s_3, r_3, \\ & & d^* \\ \mathcal{P}_{TM} & = & tran_0, (tran_1 \parallel tran_2 \parallel \ldots \parallel tran_n) \end{array}$$

Transactional memory encapsulates a set of locations. Each location i stores a value v that can be read and written. A TM algorithm description has four methods $init_t()$, $read_t(i)$, $write_t(i,v)$ and $commit_t()$. The three specific values $\mathbb{C}$, $\mathbb{A}$ and ok are returned in the description of TM algorithms to denote commitment and abortion of a transaction and normal termination of a write operation respectively. The method $init_t()$ initializes the transaction t. The method $read_t(i)$ returns the value of location i or $\mathbb{A}$ (if the transaction is aborted). The method $write_t(i,v)$ writes v to location i and returns ok (if the write is successful) or returns $\mathbb{A}$ (if the transaction is aborted). The method $commit_t()$ tries to commit transaction t and returns $\mathbb{C}$ (if the transaction is successfully committed) or returns $\mathbb{A}$ (if it is aborted). $\mathcal{P}_{TM}$ is an arbitrary client transaction. The initializing transaction $trans_0$ initializes every location to zero. It is the sequence of $init_0()$, $write_0(i,0)$ method calls for every location i and then $commit_0()$. Each transaction $trans_j$ $1 \leq j \leq n$ starts with $init_j()$ and then invokes a sequence of $read_j(i)$ and $write_j(i,v)$ method calls (for arbitrary location i and arbitrary value v). It stops invoking method calls if it receives abortion $\mathbb{A}$ from the previous method call. It finally invokes $commit_j()$ if it is not already aborted. Let Π_{TM} denote the set of transactional memory descriptions. As an example, consider the TL2 algorithm description in Figure 10. TL2 uses the strong counter *clock* to number snapshots. It reads the current snapshot number at $I01$ when a transaction starts and creates a new snapshot number at $C07$ when it wants to write back the cached values during the commit. It stores the values of locations in r registers. The value of a location is read at $R04$ and written at $C16$.

The initializing transaction $trans_0$ that initializes every location to zero is defined as follows:

$$\begin{array}{rcl} trans_0 & := & IL_0 \triangleright\ init_0(); \\ & & c_{00} \triangleright\ write_0(0,0); \\ & & c_{01} \triangleright\ write_0(1,0); \\ & & \ldots \\ & & c_{0m} \triangleright\ write_0(m,0); \\ & & CL_0 \triangleright\ commit_0() \end{array} \tag{81}$$

Each transaction $trans_j$ $1 \leq j \leq n$ is defined as follows:

$$\begin{array}{rcll} trans_j & := & IL_j \triangleright\ init_j(); & (82) \\ & & op_j & \\ op_j & := & c \triangleright\ x = read_j(v_1, v_2); & \\ & & \textbf{if}\ (\neg(x = \mathbb{A})) & \\ & & \quad op_j & \\ & | & c \triangleright\ x = write_j(v); & \\ & & \textbf{if}\ (\neg(x = \mathbb{A})) & \\ & & \quad op_j & \\ & | & CL_j \triangleright\ commit_j() & \end{array}$$

Well-formedness. The *init* method returns *ok*. The *read* method does not return *ok* or $\mathbb{C}$. The *write* method does not return $\mathbb{C}$. The *commit* method either returns $\mathbb{C}$ or $\mathbb{A}$.
$\forall c \in Returns_\pi(init)\colon arg1_\pi(c) = ok$
$\forall c \in Returns_\pi(read)\colon arg1_\pi(c) \neq ok\ \wedge\ arg1_\pi(c) \neq \mathbb{C}$
$\forall c \in Returns_\pi(write)\colon arg1_\pi(c) \neq \mathbb{C}$
$\forall c \in Returns_\pi(commit)\colon arg1_\pi(c) = \mathbb{C}\ \vee\ arg1_\pi(c) = \mathbb{A}$
In addition, it is assumed that in every execution of the transaction $trans_0$, all the *write* method calls return *ok*.

Let Π_{TM} denote the set of transactional memory specifications.

We define two functions *initOf* and *commitOf* that map a thread value to its initialization and commitment labels.

$$initOf(T) = IL_T \quad (83)$$

$$commitOf(T) = CL_T \quad (84)$$

10 Semantics

In this section, we first present a few basic lemmas about execution histories. Then, we present synchronization object types and finally we define transaction histories.

10.1 Execution histories

Lemma 10.1 (XASym). *For every execution history X and method calls l and l', if $l \prec_X l'$ then $\neg(l' \prec_X l) \wedge \neg(l' \sim_X l) \wedge \neg(l' = l)$*

Lemma 10.2 (XTrans). *For every execution history X and method calls l, l. and l'', if $l \prec_X l'$ and $l' \prec l''$ then $l \prec_X l''$*

Lemma 10.3 (XXTrans). *For every execution history X and method calls l_1, l_2, l_3, and l_4, if $l_1 \prec_X l_2$, $l_2 \precsim_X l_3$, and $l_3 \prec_X l_4$ then $l_1 \prec_X l_4$*

Lemma 10.4 (XTotal). *For every execution history X and method calls l and l', if $l \in X$, and $l' \in X$, then $(l \prec_X l') \vee (l' \prec_X l) \vee (l \sim_X l') \vee (l = l')$*

Lemma 10.5 (X2X). *For every execution history X and method calls l and l', if $l \prec_X l'$ then $l \in X$, and $l' \in X$.*

Lemma 10.6 (XI2X). *For every execution history X and method calls l, l', and l'' if $l \prec_X l'$ and $inv(l') \lhd_X inv(l'')$ then $l \prec_X l''$.*

Lemma 10.7 (RX2X). *For every execution history X and method calls l, l', and l'' if $ret(l) \lhd_X ret(l')$ and $l' \prec_X l''$ then $l \prec_X l''$.*

10.2 Synchronization Object Types

In this subsection, we define the semantics of basic and linearizable objects. Then, we define the interface and the sequential specifications of the following abstract object types: register, lock, try-lock, counter, set and map. For each abstract object type, we define concrete synchronization object types. We define the following synchronization object types: basic register, atomic register, atomic cas register, lock, try-lock, strong counter, basic set and basic map. For each synchronization object type, we present lemmas that characterize the properties of its execution histories. Please see Section 15.1.2 for notes on the proof of the lemmas that we present in this subsection.[1]

Basic, Sequentially-consistent and Linearizable Object Types

The abstract type of each object o specifies the sequential specification of o, denoted by $SeqSpec(o)$, that is the prefix-closed set of correct sequential histories of o. In the following subsections, we will consider several synchronization object types and define their sequential specifications.

We consider three concurrent types: basic, sequentially-consistent and linearizable. Sequentially-consistent and linearizable objects comply with their sequential specification in every concurrent execution. Basic objects, on the other hand, comply with their sequential specification if they are accessed sequentially.

Definition 10.8 (Basic Object Semantics)**.** Every sequential execution on a basic object is an execution in its sequential specification. The semantics of a basic object o, $\mathbb{H}_B(o)$, is a set of histories that is constrained as follows:

$$\mathbb{H}_B(o) \cap Sequential \subseteq SeqSpec(o) \tag{85}$$

Definition 10.9 (Sequentially-consistent Object Semantics)**.** An execution history X is sequentially-consistent for an object o iff there is an indistinguishable sequential history L that is in the sequential specification of o. L is a sequentialization and $\prec_L$ is a sequentialization order of X. The semantics of a sequentially-consistent object o, $\mathbb{H}_L(o)$, is defined as the following set of execution and sequentialization pairs.

$$\mathbb{H}_L(o) = \{(X, L) \mid X \equiv L \ \wedge \ L \in SeqSpec(o) \ \wedge \ \forall T \in X\colon \ \prec_{X|T} \subseteq \prec_L\} \tag{86}$$

Note that the notion of sequential consistency defined above is for operations on a single object in contrast to sequential consistency for operations on multiple objects. The notion defined above is also called cache coherence.

Definition 10.10 (Linearizable Object Semantics)**.** An execution history X is linearizable for an object o iff there is an indistinguishable sequential history L that is in the sequential specification of o and is real-time-preserving. L is a linearization and $\prec_L$ is a linearization order of X. The semantics of a linearizable object o, $\mathbb{H}_L(o)$, is defined as the following set of execution and linearization pairs.

$$\mathbb{H}_L(o) = \{(X, L) \mid X \equiv L \ \wedge \ L \in SeqSpec(o) \ \wedge \ \prec_X \subseteq \prec_L\} \tag{87}$$

Note that sequentially-consistent objects preserve execution order of method calls in the justifying sequential order only within threads while linearizable objects preserve it even across threads.

We now present lemmas for serialization and linearization orders.

Lemma 10.11 (X2L)**.** *For every linearization L of an execution history X on object o and method calls l and l', if $l \prec_X l'$ then $l \prec_L l'$.*

Lemma 10.12 (X2L')**.** *For every linearization L of an execution history X on object o and method calls l and l', if $l \prec_L l'$ then $l \precsim_X l'$.*

Lemma 10.13 (LASym)**.** *For every sequentialization or linearization L of an execution history X on object o and method calls l and l', if $l \prec_L l'$ then $\neg(l' \prec_L l) \ \wedge \ \neg(l = l')$*

Lemma 10.14 (LTrans)**.** *For every sequentialization or linearization L of an execution history X on object o and method calls l, l', and l'', if $l \prec_L l'$ and $l' \prec_L l''$ then $l \prec_L l''$.*

Lemma 10.15 (LTotal)**.** *For every sequentialization or linearization L of an execution history X on object o and method calls l and l', if $l \in X$ and $l' \in X$ then $(l \prec_L l') \ \vee \ (l' \prec_L l) \ \vee \ (l = l')$*

Lemma 10.16 (L2X)**.** *For every sequentialization or linearization L of an execution history X on object o and method calls l and l', if $(l \prec_L l')$ then $l \in X$, $l' \in X$, and l and l' are both on o.*

[1] In this subsection, we use $\forall$ and $\exists$ as a notational convenience. $\forall l\colon p$ can be rewritten as $\bigwedge_{(l \in Labels(X))} p(X)$ and $\exists l\colon p$ can be rewritten as $\bigvee_{(l \in Labels(X))} p(X)$.

Lemma 10.17 (XLTrans)**.** *For every linearization L of an execution history X on object o and method calls l_1, l_2, l_3, and l_4, if $l_1 \prec_X l_2$, $l_2 \prec_L l_3$, $l_3 \prec_X l_4$, then $l_1 \prec_X l_4$*

See section 15.1.2 for proofs.

10.2.1 Register

Register. A register *reg* is an object that encapsulates a value and supports *read* and *write* methods. The method call $reg.read()$ returns the current encapsulated value of *reg*. The method call $reg.write(v)$ overwrites the encapsulated value of *reg* with v.

Definition 10.18. The sequential specification of register *reg* is the set of sequential histories of *read* and *write* method calls on *reg* where every read returns the argument of the latest preceding write (regardless of thread identifiers). (Note that it is assumed that a write method call initializes the register before other methods are invoked.) The sequential specification of a register r, $SeqSpec(r)$, is defined as follows:

$$isXRead_{X,r}(l_R) = l_R \in X \ \wedge \ obj_X(l_R) = r \ \wedge \ name_X(l_R) = read \tag{88}$$

$$isXWrite_{X,r}(l_W) = l_W \in X \ \wedge \ obj_X(l_W) = r \ \wedge \ name_X(l_W) = write \tag{89}$$

$$NoWriteBetween_{X,r}(l_W, l_R) = \forall l'_W : isXWrite_{X,r}(l'_W) \Rightarrow (l'_W \preceq_X l_W \ \vee \ l_R \prec_X l'_W) \tag{90}$$

$$isXWriter_{X,r}(l_W, l_R) = isXWrite_{X,r}(l_W) \ \wedge \ l_W \prec_X l_R \ \wedge \ NoWriteBetween_{X,r}(l_W, l_R) \tag{91}$$

$$Legal(r) = \{S \mid \forall l_R : isXRead_{S,r}(l_R) \Rightarrow \exists l_W : isXWriter_{S,r}(l_W, l_R) \ \wedge \ retv_S(l_R) = arg1_S(l_W)\} \tag{92}$$

$$SeqSpec(r) = \{S \mid S|r = S \ \wedge \ S \in Sequential \cap Legal(r)\} \tag{93}$$

Basic Register. A basic register is a basic instance of the register type.

Let *BasicRegister* denote the type of basic registers.

Lemma 10.19. *In every sequential execution on a basic register, every read reads the value that the latest preceding write writes. Formally,*

$$\forall reg \in BasicRegister : \forall X \in \mathbb{H}_B(reg) : X \in Sequential \Rightarrow \forall l_R : isXRead_{X,reg}(l_R) \Rightarrow \exists l_W : isXWriter_{X,reg}(l_W, l_R) \ \wedge \ retv_X(l_R) = arg1_X(l_W) \tag{94}$$

Two concurrent read method calls on a register do not conflict. Thus, basic registers can maintain consistency even when the execution involves concurrent read method calls. Let us define

$$isXRaceFree_{X,r}(l) = \forall l_w : isXWrite_{X,r}(l_w) \Rightarrow l_w \preceq_X l \ \vee \ l \prec_X l_w \tag{95}$$

$$isXSequentiallyWritten_r(X) = \forall l \in X : isXWrite_{X,r}(l) \Rightarrow isRaceFree_{X,r}(l) \tag{96}$$

A method call is race-free if an only if there is no write method call that executes concurrent to it. An execution is sequentially-written if and only if every pair of write method calls on it are ordered in the execution order or in other words, every write method call on it is race-free.

Definition 10.20 (Basic Register Semantics)**.** An execution history on a basic register is in the semantics of the basic register if and only if it is not sequentially-written or it is sequentially-written and every race-free read reads the value that the latest preceding write writes. The semantics of a basic register r, $\mathbb{H}_B(r)$, is defined as follows.

$$\mathbb{H}_B(r) = \{X \mid X|o = X \ \wedge \ isXSequentiallyWritten_r(X) \Rightarrow \forall l_r : isXRead_{X,r}(l_r) \ \wedge \ isXRaceFree_{X,r}(l_r) \Rightarrow \exists l_w : isXWriter_{X,r}(l_w, l_r) \ \wedge \ retv_X(l_r) = arg1_X(l_w)\} \tag{97}$$

Note that if an execution is not sequentially-written, reads may return arbitrary values. Similarly, racy reads may return arbitrary values.

Note that this definition satisfies the constraint of Definition 10.8.

Note that basic register models Lamport's notion of safe register [26].

Lemma 10.21 (BReg). *In every sequentially-written execution on a basic register, every race-free read reads the value that the latest preceding write writes. Formally,*

$$\begin{aligned}&\forall reg \in BasicRegister: \forall X \in \mathbb{H}_B(reg): isXSequentiallyWritten_r(X) \Rightarrow \\ &\forall l_R: isXRead_{X,reg}(l_R) \ \wedge \ isXRaceFree_{X,r}(l_R) \Rightarrow \\ &\exists l_W: isXWriter_{X,reg}(l_W, l_R) \ \wedge \\ &\quad ret v_X(l_R) = arg1_X(l_W)\end{aligned} \tag{98}$$

Atomic Register. An atomic register is a linearizable instance of the register type.

Let *AtomicRegister* denote the type of atomic registers.

Let us define

$$LNoWriteBetween_{X,L,r}(l_W, l_R) = \forall l'_W: isXWrite_{X,r}(l'_W) \Rightarrow (l'_W \preceq_L l_W \ \vee \ l_R \prec_L l'_W) \tag{99}$$

$$\begin{aligned}isLWriter_{X,L,r}(l_W, l_R) = \ & isXWrite_{X,r}(l_W) \ \wedge \\ & l_W \prec_L l_R \ \wedge \\ & LNoWriteBetween_{X,L,r}(l_W, l_R)\end{aligned} \tag{100}$$

Lemma 10.22 (AReg). *In every execution on an atomic register, every read reads the value written by the last write linearized before it. Formally,*

$$\begin{aligned}&\forall r \in AtomicRegister: \forall (X, L) \in \mathbb{H}_L(r): \\ &\forall l_R: isXRead_{X,r}(l_R) \Rightarrow \\ &\exists l_W: isLWriter_{X,L,r}(l_W, l_R) \ \wedge \\ &\quad ret v_X(l_R) = arg1_X(l_W)\end{aligned} \tag{101}$$

Sequentially-consistent Register. A sequentially-consistent register is a sequentially-consistent instance of the register type.

Let *SCRegister* denote the type of sequentially-consistent registers.

Consider the following four concurrent threads.

T_1	T_2	T_3	T_4
$L_{11} \triangleright \ r_1.write(1)$ $\parallel$	$L_{21} \triangleright \ r_2.write(1)$ $\parallel$	$L_{31} \triangleright \ x_1 = r_1.read()$ $\parallel$	$L_{41} \triangleright \ y_2 = r_2.read()$
		$L_{32} \triangleright \ y_1 = r_2.read()$	$L_{42} \triangleright \ x_2 = r_1.read()$
		$\{L_{31} \rightarrow L_{32}\}$	$\{L_{41} \rightarrow L_{42}\}$

If r_1 and r_2 are sequentially-consistent registers, there is an execution that results in the following values for the variables: $x_1 = 1$, $y_1 = 0$, $y_2 = 1$ and $x_2 = 0$.

These values can be justified by the sequentialization order

(1) $L_{r_1} = L_{42} \triangleright \ x_2 = r_1.read() \ \cdot \ L_{11} \triangleright \ r_1.write(1) \ \cdot \ L_{31} \triangleright \ x_1 = r_1.read()$

for r_1 and the sequentialization order

(2) $L_{r_2} = L_{32} \triangleright \ y_1 = r_2.read() \ \cdot \ L_{21} \triangleright \ r_2.write(1) \ \cdot \ L_{41} \triangleright \ y_2 = r_2.read()$

for r_2.

If r_1 and r_2 are atomic registers, there is no execution that results in the values above for the variables. The real-time-preservation property precludes these executions. We assume that there is such an execution and show a contradiction. To have the above values for the variables, the linearization order of r_1 and r_2 should be as above in 1 and 2. By the program orders above, we have (3) $L_{31} \prec_X L_{32}$ (4) $L_{41} \prec_X L_{42}$. By X2L' on 2, we have (5) $L_{32} \preceq_X L_{41}$. By XXTrans on 3, 5 and 4, we have (6) $L_{31} \prec_X L_{42}$. By X2L on 6, we have $L_{31} \prec_{r_1} L_{42}$ that contradicts 1.

10.2.2 CAS (Compare-And-Swap) Register

A CAS register is an object that encapsulates a value and supports the *cas* method in addition to *read* and *write* methods. The method call $r.cas(v_1, v_2)$ updates the value of the register to v_2 and returns *true* if the current value of the register is v_1. It returns $false$ otherwise.

A *successful write* is either a *write* method call or a successful *cas* method call. The *written value* of a successful write is its first argument, if it is a *write* method call or is its second argument, if it is a *cas* method call.

Definition 10.23. The sequential specification of cas register *reg* is the set of sequential histories of *read*, *write* and *cas* method calls on *reg* with the following two conditions. Every *read* returns the written value of the latest preceding successful write (regardless of thread identifiers). (Note that it is assumed that a write method call initializes the register before other methods are invoked.) Every *cas* with the first argument v_1 returns *true* if the written value of the latest preceding successful write is v_1 and returns $false$ otherwise.

Atomic CAS Register. An atomic CAS register is a linearizable instance of CAS register type.

Let *AtomicCASRegister* denote the type of Atomic CAS registers.

Let us define

$$isXCAS_{X,r}(l_W) = l_W \in X \ \wedge \ obj_X(l_W) = r \ \wedge \ name_X(l_W) = cas \tag{102}$$

$$isXCWrite_{X,r}(l_W) = isXWrite(l_W) \ \vee \ (isXCAS(l_W) \ \wedge \ retv_X(l_W) = true) \tag{103}$$

$$writtenValue_X(l_W) = \begin{cases} arg1_X(l_W) & \text{if } name_X(l_W) = write \\ arg2_X(l_W) & \text{if } name_X(l_W) = cas \end{cases} \tag{104}$$

$$LNoWriteBetween_{X,L,r}(l_W, l_R) = \forall l'_W : isXCWrite_{X,r}(l'_W) \Rightarrow (l'_W \preceq_L l_W \ \vee \ l_R \prec_L l'_W) \tag{105}$$

$$\begin{aligned} isLCWriter_{X,L,r}(l_W, l_R) = \ & isXCWrite_{X,r}(l_W) \ \wedge \\ & l_W \prec_L l_R \ \wedge \\ & LNoWriteBetween_{X,L,r}(l_W, l_R) \end{aligned} \tag{106}$$

Lemma 10.24 (CASRegRead)**.** *In every execution on an atomic cas register, every read returns the value the last successful write linearized before it writes. Formally,*

$$\begin{aligned} & \forall r \in AtomicCASRegister : \forall (X, L) \in \mathbb{H}_L(r) : \\ & \forall l_R : isXRead_{X,r}(l_R) \Rightarrow \\ & \exists l_W : isLCWriter_{X,L,r}(l_W, l_R) \ \wedge \\ & \qquad retv_X(l_R) = arg1_X(l_W) \end{aligned} \tag{107}$$

Lemma 10.25 (CASRegCAS)**.** *In every execution on an atomic cas register, every cas returns true if its first argument is equal to the argument of the last successful write linearized before it and returns false otherwise. Formally,*

$$\begin{aligned} & \forall reg \in AtomicCASRegister : \forall (X, Reg) \in \mathbb{H}_L(reg) : \\ & \forall l_C, l_W : \\ & \quad isXCAS_{X,reg}(l_C) \ \wedge \\ & \quad isLCWriter_{X,Reg,reg}(l_W, l_R) \\ & \Rightarrow \\ & \quad (writtenValue_X(l_W) = arg1_X(l_C) \Rightarrow retv_X(l_C) = true) \ \wedge \\ & \quad (\neg(writtenValue_X(l_W) = arg1_X(l_C)) \Rightarrow retv_X(l_C) = false) \end{aligned} \tag{108}$$

10.2.3 Lock

Abstract lock. An abstract lock l is an object that encapsulates a state, acquired $\mathbb{A}$ or released $\mathbb{R}$, and supports the following methods: *lock*: The method call $l.lock()$ changes the state from $\mathbb{R}$ to $\mathbb{A}$. *unlock*: The method call $l.unlock()$ changes the state from $\mathbb{A}$ to $\mathbb{R}$. *read*: The method call $l.read()$ returns *true* if the state of *lock* is $\mathbb{A}$ and $false$ otherwise. The method calls *lock* and *unlock* are mutating method calls. The method call *read* is an accessor method call.

Definition 10.26. The sequential specification of a lock l is the set of sequential histories L of *lock*, *unlock*, and *read* method calls on l where the sub-history of L for mutating methods is an alternating sequence of *lock* and *unlock* methods and every *read* method call in L returns *true* if the last mutating method call before it in L is a *lock* and returns $false$ otherwise.

Lock. A lock is a linearizable instance of the abstract lock type.

Let *Lock* denote the type of locks.

Now, we present some preliminary definitions and then lemmas about locks.

$$isXLock_{X,lo}(l) = \quad (109)$$
$$l \in X \ \wedge \ obj_X(l) = lo \ \wedge \ name_X(l) = lock$$
$$isXUnlock_{X,lo}(l) = \quad (110)$$
$$l \in X \ \wedge \ obj_X(l) = lo \ \wedge \ name_X(l) = unlock$$
$$isXRead_{X,lo}(l) = \quad (111)$$
$$l \in X \ \wedge \ obj_X(l) = lo \ \wedge \ name_X(l) = read$$

The common usage protocol for locks is that a thread unlocks a lock only if it has already acquired it. Many languages including Java enforce this property of programs by runtime checks. We capture this property as follows.

Definition 10.27. A history is owner-respecting for a lock if every thread in the history releases the lock only after it has already acquired it.

$$\begin{aligned} &isXOwnerRespecting_{lo}(X) = \quad (112)\\ &\quad \forall l : isXUnlock_{X,lo}(l) \Rightarrow \\ &\quad \exists l' : isXLock_{X,lo}(l') \wedge \\ &\qquad thread_X(l') = thread_X(l) \wedge \\ &\qquad l' \prec_X l \wedge \\ &\qquad \forall l'' : (isXUnLock_{X,lo}(l'') \ \wedge \ thread_X(l'') = thread_X(l)) \Rightarrow (l'' \prec_X l' \ \vee \ l \preceq_X l'') \end{aligned}$$

Lemma 10.28. *If l is a lock, X is an owner-respecting history of l and L is the linearization of X, then the sub-history of L for mutating method calls is a sequence of pairs of lock and unlock method calls by the same thread (possibly followed by a lock method call).*

Lemma 10.29 (Lock)**.** *In an owner-respecting execution for a lock l, if a lock method call by a thread T_1 is linearized before an unlock method call by a thread T_2, then an unlock method call by T_1 is linearized before a lock method call by T_2. Formally,*

$$\begin{aligned} &\forall o \in Lock : \forall (X, L) \in \mathbb{H}_L(o) : \forall l_{l1}, l_{u2} : \quad (113)\\ &\quad (isXOwnerRespecting_o(X) \ \wedge \\ &\quad isXLock_{X,o}(l_{l1}) \ \wedge \\ &\quad isXUnlock_{X,o}(l_{u2}) \ \wedge \\ &\quad l_{l1} \prec_L l_{u2}) \Rightarrow \\ &\exists l_{u1}, l_{l2} : \\ &\quad isXUnlock_{X,o}(l_{u1}) \ \wedge \ thread_X(l_{l1}) = thread_X(l_{u1}) \ \wedge \\ &\quad isXLock_{X,o}(l_{l2}) \ \wedge \ thread_X(l_{l2}) = thread_X(l_{u2}) \ \wedge \\ &\quad l_{u1} \prec_L l_{l2} \end{aligned}$$

Lemma 10.30 (LockReadL)**.** *In an owner-respecting execution for a lock l, if a read method call that returns $false$ is linearized before an unlock method call by a thread T, then the read method call is linearized before a lock method call by T. Formally,*

$$\begin{aligned} &\forall o \in Lock : \forall (X, L) \in \mathbb{H}_L(o) : \forall l_{u1}, l_{r2} : \quad (114)\\ &\quad (isXOwnerRespecting_o(X) \ \wedge \\ &\quad isXRead_{X,o}(l_{r2}) \ \wedge \ retv_X(l_{r2}) = false \\ &\quad isXUnlock_{X,o}(l_{u1}) \ \wedge \\ &\quad l_{r2} \prec_L l_{u1}) \Rightarrow \\ &\exists l_{l1} : \\ &\quad isXLock_{X,o}(l_{l1}) \ \wedge \ thread_X(l_{l1}) = thread_X(l_{u1}) \ \wedge \\ &\quad l_{r2} \prec_L l_{l1} \end{aligned}$$

Lemma 10.31 (LockReadR)**.** *In an owner-respecting execution for a lock l, if a lock method call by a thread T is linearized before a read method call that returns $false$, then an unlock method call by T is linearized before the read method call. Formally,*

$$\begin{aligned}
&\forall o \in Lock\colon \forall (X,L) \in \mathbb{H}_L(o)\colon \forall l_{l1}, l_{r2}\colon \\
&\quad (isXOwnerRespecting_o(X) \ \wedge \\
&\quad isXLock_{X,o}(l_{l1}) \ \wedge \\
&\quad isXRead_{X,o}(l_{r2}) \ \wedge \ retv_X(l_{r2}) = false \\
&\quad l_{l1} \prec_L l_{r2}) \Rightarrow \\
&\exists l_{u1}\colon \\
&\quad isXUnlock_{X,o}(l_{u1}) \ \wedge \ thread_X(l_{l1}) = thread_X(l_{u1}) \ \wedge \\
&\quad l_{u1} \prec_L l_{r2}
\end{aligned} \tag{115}$$

Lemma 10.32 (LockReadM)**.** *In an owner-respecting execution for a lock l, every read method call that is linearized between a pair of matching lock and unlock method calls returns true. Formally,*

$$\begin{aligned}
&\forall o \in Lock\colon \forall (X,L) \in \mathbb{H}_L(o)\colon \forall l_{l1}, l_{u1}, l_{r2}\colon \\
&\quad (isXOwnerRespecting_o(X) \ \wedge \\
&\quad isXLock_{X,o}(l_{l1}) \ \wedge \\
&\quad isXUnlock_{X,o}(l_{u1}) \ \wedge \\
&\quad thread_X(l_{l1}) = thread_X(l_{u1}) \ \wedge \\
&\quad \forall l'_{u1}\colon (isXUnlock_{X,o}(l'_{u1}) \ \wedge \ thread_X(l_{l1}) = thread_X(l'_{u1})) \Rightarrow (l'_{u1} \prec_X l_{l1} \ \vee \ l_{u1} \preceq_X l'_{u1}) \\
&\quad isXRead_{X,o}(l_{r2}) \ \wedge \\
&\quad l_{l1} \prec_L l_{r2} \ \wedge \ l_{r2} \prec_L l_{u1}) \\
&\Rightarrow \\
&\quad retv_X(l_{r2}) = true
\end{aligned} \tag{116}$$

10.2.4 Try-lock

Abstract Try-lock. A try-lock l is an object that encapsulates an abstract state, acquired $\mathbb{A}$ or released $\mathbb{R}$, and in addition to *lock*, *unlock* and *read* methods, it supports the *trylock* method. If the state of the *lock* is $\mathbb{R}$, $l.trylock()$ changes it to $\mathbb{A}$ and returns *true*. Otherwise, it returns $false$.

We call a *lock* method call or a successful *tryLock* method call, a *successful lock* method call. We call a *lock* method call, successful *tryLock* method call or *unlock* method call, a *mutating* method call.

Definition 10.33. The sequential specification of a try-lock l is the set of sequential histories L of *lock*, *unlock*, *read* and *tryLock* method calls on l with the following conditions: The last mutating method call before a successful lock method call is an unlock method call. Similarly, the last mutating method call before an unlock method call is a successful lock method call. A *tryLock* method call returns *true* if the latest preceding mutating method call is an *unlock* and returns $false$ otherwise. Similarly, A *read* method call returns *true* if the latest preceding mutating method call is a successful lock and returns $false$ otherwise.

Try-Lock. A try-lock is a linearizable instance of the abstract try-lock type.

Let $TryLock$ denote the type of try-locks.

Similar to the *Lock* type, after some preliminary definitions, we define the owner-respecting histories and state the $TryLock$ type lemmas.

$$\begin{aligned}
&isXTryLock_{X,o}(l) = \\
&\quad l \in X \ \wedge \ obj_X(l) = o \ \wedge \ name_X(l) = tryLock
\end{aligned} \tag{117}$$

$$\begin{aligned}
&isXTLock_{X,o}(l) = \\
&\quad isXLock_{X,o}(l) \ \vee \ (isXTryLock_{X,o}(l) \ \wedge \ retv_X(l) = true)
\end{aligned} \tag{118}$$

The intuition for owner-respecting histories remains the same. A history is owner-respecting for a try-lock if every thread in the history releases the lock only after it has already acquired it. The minor difference from the prior definition for locks is

that the acquisition of a try-lock is either by a *lock* method call or a successful *tryLock* method call.

$$
\begin{aligned}
&isXTOwnerRespecting_o(X) = \\
&\quad \forall l : isXUnlock_{X,o}(l) \Rightarrow \\
&\quad \exists l' : isXTLock_{X,o}(l') \wedge \\
&\qquad thread_X(l') = thread_X(l) \wedge \\
&\qquad l' \prec_X l \wedge \\
&\qquad \forall l'' : (isXUnLock_{X,o}(l'') \wedge thread_X(l'') = thread_X(l)) \Rightarrow l'' \prec_X l' \vee l \preceq_X l''
\end{aligned}
\tag{119}
$$

Lemma 10.34. *If l is a try-lock, X is an owner-respecting history of l and L is the linearization of X, then the sub-history of L for mutating method calls is a sequence of pairs of successful lock and unlock method calls by the same thread (possibly followed by a successful lock method call).*

Lemma 10.35 (TryLock)**.** *In an owner-respecting execution for a try-lock l, if a successful lock method call by a thread T_1 is linearized before an unlock method call by a thread T_2, then an unlock method call by T_1 is linearized before a successful lock method call by T_2. Formally,*

$$
\begin{aligned}
&\forall o \in TryLock : \forall (X, L) \in \mathbb{H}_L(o) : \forall l_{l1}, l_{u2} : \\
&\quad (isXTOwnerRespecting_o(X) \wedge \\
&\quad isXTLock_{X,o}(l_{l1}) \wedge \\
&\quad isXUnlock_{X,o}(l_{u2}) \wedge \\
&\quad l_{l1} \prec_L l_{u2}) \Rightarrow \\
&\exists l_{u1}, l_{l2} : \\
&\quad isXUnlock_{X,o}(l_{u1}) \wedge thread_X(l_{l1}) = thread_X(l_{u1}) \wedge \\
&\quad isXTLock_{X,o}(l_{l2}) \wedge thread_X(l_{l2}) = thread_X(l_{u2}) \wedge \\
&\quad l_{u1} \prec_L l_{l2}
\end{aligned}
\tag{120}
$$

Lemma 10.36 (TryLockReadL)**.** *In an owner-respecting execution for a try-lock l, a read method call that returns false is linearized before if an unlock method call by a thread T then the read method call is linearized before a successful lock method call by T. Formally,*

$$
\begin{aligned}
&\forall o \in TryLock : \forall (X, L) \in \mathbb{H}_L(o) : \forall l_{u1}, l_{r2} : \\
&\quad (isXTOwnerRespecting_o(X) \wedge \\
&\quad isXRead_{X,o}(l_{r2}) \wedge retv_X(l_{r2}) = false \\
&\quad isXUnlock_{X,o}(l_{u1}) \wedge \\
&\quad l_{r2} \prec_L l_{u1}) \Rightarrow \\
&\exists l_{l1} : \\
&\quad isXTLock_{X,o}(l_{l1}) \wedge thread_X(l_{l1}) = thread_X(l_{u1}) \wedge \\
&\quad l_{r2} \prec_L l_{l1}
\end{aligned}
\tag{121}
$$

Lemma 10.37 (TryLockReadR)**.** *In an owner-respecting execution for a try-lock l, if a successful lock method call by a thread T is linearized before a read method call that returns false, then an unlock method call by T is linearized before the read method call. Formally,*

$$
\begin{aligned}
&\forall o \in TryLock : \forall (X, L) \in \mathbb{H}_L(o) : \forall l_{l1}, l_{r2} : \\
&\quad (isXTOwnerRespecting_o(X) \wedge \\
&\quad isXTLock_{X,o}(l_{l1}) \wedge \\
&\quad isXRead_{X,o}(l_{r2}) \wedge retv_X(l_{r2}) = false \\
&\quad l_{l1} \prec_L l_{r2}) \Rightarrow \\
&\exists l_{u1} : \\
&\quad isXUnlock_{X,o}(l_{u1}) \wedge thread_X(l_{l1}) = thread_X(l_{u1}) \wedge \\
&\quad l_{u1} \prec_L l_{r2}
\end{aligned}
\tag{122}
$$

Lemma 10.38 (TryLockReadM). *In an owner-respecting execution for a try-lock l, every read method call that is linearized between a pair of matching successful and unlock method calls returns true. Formally,*

$$
\begin{aligned}
&\forall o \in TryLock\colon \forall (X,L) \in \mathbb{H}_L(o)\colon \forall l_{l1}, l_{u1}, l_{r2}\colon \\
&\quad (isXOwnerRespecting_o(X) \ \wedge \\
&\quad isXTLock_{X,o}(l_{l1}) \ \wedge \\
&\quad isXUnlock_{X,o}(l_{u1}) \ \wedge \\
&\quad thread_X(l_{l1}) = thread_X(l_{u1}) \ \wedge \\
&\quad \forall l'_{u1}\colon (isXUnlock_{X,o}(l'_{u1}) \ \wedge \ thread_X(l_{l1}) = thread_X(l'_{u1})) \Rightarrow (l'_{u1} \prec_X l_{l1} \ \vee \ l_{u1} \preceq_X l'_{u1}) \\
&\quad isXRead_{X,o}(l_{r2}) \ \wedge \\
&\quad l_{l1} \prec_L l_{r2} \ \wedge \ l_{r2} \prec_L l_{u1}) \\
&\Rightarrow \\
&\quad retv_X(l_{r2}) = true
\end{aligned}
\tag{123}
$$

10.2.5 Sequence-lock

Abstract seq-lock. A seq-lock l is an object that encapsulates a number and an abstract state, acquired $\mathbb{A}$ or released $\mathbb{R}$. It supports the *read*, *compareAndLock* and *incAndUnlock* methods. The method call $l.read()$ returns the pair of the encapsulated number and *true* if the state of *lock* is $\mathbb{A}$ and $false$ otherwise. The method call $l.compareAndLock(n)$ compares the encapsulated number with n and if they are equal, changes the state from $\mathbb{R}$ to $\mathbb{A}$ and returns *true*. Otherwise, it does not change the state of the seq-lock and returns $false$. The method call $l.incAndUnlock()$ increments the encapsulated number and changes the state from $\mathbb{A}$ to $\mathbb{R}$.

A successful *compareAndLock* and *incAndUnlock* are mutating method calls. The method call *read* is an accessor method call.

Definition 10.39. The sequential specification of a seq-lock l is the set of sequential histories L of *read*, *compareAndLock*, and *incAndUnlock* method calls on l with the following conditions:

Every *read* method call returns the pair of the number of *incAndUnlock* method calls before it and *true* if the last mutating method call before it is a successful *compareAndLock* and $false$ otherwise.

A *compareAndLock* method call returns *true* if the last mutating method call before it is an *incAndUnlock* method call and the number of *incAndUnlock* method calls before it is equal to its argument. It returns $false$ otherwise.

The last mutating method call before an *incAndUnlock* method call is a successful *compareAndLock* method call.

Seq-Lock. A seq-lock is a linearizable instance of the abstract seq-lock type.

Let *SeqLock* denote the type of seq-locks.

10.2.6 Counter

Abstract Counter: A counter c is an object that encapsulates a number and supports the following two methods: The method call $c.read()$ returns the current value of c. The method call $c.iaf()$ increments the value of c and returns the incremented value.

Definition 10.40. The sequential specification of a counter c is the set of sequential histories of *read* and *iaf* method calls on c where every method call returns the number of *iaf* method calls before it (including the method call itself). Note that it is assumed that the initial value of the counter is zero.

Strong Counter. A strong counter is a linearizable instance of abstract counter type.

Let *SCounter* denote the type of strong counters.

Lemma 10.41 (SCounter). *The return value of every method call that is linearized before an iaf method call is smaller than the return value of the iaf method call. Formally,*

$$\begin{aligned}&\forall c \in SCounter : \forall (X, C) \in \mathbb{H}_L(c) : \forall l, l' : \\ &\quad l \in X \ \wedge \ l' \in X \ \wedge \ name_X(l') = iaf \ \wedge \ l \prec_C l' \\ &\Rightarrow \\ &\quad retv_X(l) < retv_X(l')\end{aligned} \tag{124}$$

10.2.7 Set

A set s is an object that represents a set of values and supports the following methods: *add*: The method call $s.add(v)$ adds value v to set s. *contains*: The method call $s.contains(v)$ returns $true$ if v is a member of s and $false$ otherwise.

Definition 10.42. The sequential specification of a set s is the set of sequential histories of *add* and *contains* method calls on s where every *contains* method call returns $true$ if there is a preceding *add* method call with the same argument, and returns $false$ otherwise. Note that it is assumed that the set is initially empty.

Basic Set. A basic set is a basic instance of set type.

Let *BasicSet* denote the type of basic sets.

Let us define

$$\begin{aligned}&isXContains_{X,s}(l) = \\ &\quad l \in X \ \wedge \ obj_X(l) = s \ \wedge \ name_X(l) = contains\end{aligned} \tag{125}$$

$$\begin{aligned}&isXAdd_{X,s}(l) = \\ &\quad l \in X \ \wedge \ obj_X(l) = s \ \wedge \ name_X(l) = add\end{aligned} \tag{126}$$

Lemma 10.43 (BasicSetContains). *In every sequential execution on a basic set, for every contains method call that returns true, there is a preceding add method call with the same argument. Formally,*

$$\begin{aligned}&\forall s \in BasicSet : \forall X \in \mathbb{H}_B(s) : X \in Sequential \Rightarrow \\ &\forall l_c : isXContains_{X,s}(l_c) \ \wedge \ retv_X(l_c) = true \Rightarrow \\ &\exists l_a : isXAdd_{X,s}(l_a) \ \wedge \\ &\quad arg1(l_a) = arg1(l_c) \ \wedge \ l_a \prec_X l_c\end{aligned} \tag{127}$$

Lemma 10.44 (BasicSetAdd). *In every sequential execution on a basic set, every contains method call that succeeds an add method call with the same argument returns true. Formally,*

$$\begin{aligned}&\forall s \in BasicSet : \forall X \in \mathbb{H}_B(s) : X \in Sequential \Rightarrow \\ &\forall l_c, l_a : \\ &\quad isXContains_{X,s}(l_c) \ \wedge \\ &\quad isXAdd_{X,s}(l_a) \ \wedge \\ &\quad arg1(l_a) = arg1(l_c) \ \wedge \ l_a \prec_X l_c \\ &\Rightarrow \\ &\quad retv_X(l_c) = true\end{aligned} \tag{128}$$

10.2.8 Map

A map m is an object that represents a mapping from a set of keys to a set of values and supports the following methods: *put*: The method call $m.put(k, v)$ adds or updates the mapping of the key k to the value v ($v \neq \perp$) in the map m. *get*: The method call $m.get(k)$ returns the value that the map m associates with the key k. It returns $\perp$ if m does not map k.

Definition 10.45. The sequential specification of a map m is the set of sequential histories of *put* and *get* method calls on m where every *get* method call returns $\perp$ if there is no preceding *put* method call with the same key argument; otherwise it returns the second argument of the latest preceding *put* method call with the same key argument. Note that it is assumed that the map is initially empty.

Basic Map. A basic set is a basic instance of map type.

Let *BasicMap* denote the type of basic maps.

Let us define

$$isXGet_{X,m}(l) = \tag{129}$$
$$\quad l \in X \ \wedge \ obj_X(l) = m \ \wedge \ name_X(l) = get$$
$$isXPut_{X,m}(l) = \tag{130}$$
$$\quad l \in X \ \wedge \ obj_X(l) = m \ \wedge \ name_X(l) = put$$
$$isXPutter_{X,m}(l_p, l_g) \Leftrightarrow \tag{131}$$
$$\quad isXPut_{X,m}(l_p) \ \wedge \ arg1_X(l_p) = arg1_X(l_g) \ \wedge \ l_p \prec_X l_g \ \wedge \tag{132}$$
$$\quad \forall l'_p \colon isXPut_{X,m}(l'_p) \ \wedge \ arg1_X(l'_p) = arg1_X(l_g) \Rightarrow (l'_p \preceq_X l_p \ \vee \ l_g \prec_X l'_p) \tag{133}$$

Lemma 10.46 (BasicMapGet)**.** *In every sequential execution on a basic map, the return value of every get method call that does not return $\perp$ is equal to the value argument of the latest preceding put method call with the same key argument. Formally,*

$$\forall m \in BasicMap \colon \forall X \in \mathbb{H}_B(m) \colon X \in Sequential \Rightarrow \tag{134}$$
$$\forall l_g \colon isXGet_{X,m}(l_g) \ \wedge \ \neg(retv_X(l_g) = \perp) \Rightarrow$$
$$\exists l_p \colon isPutter_{X,m}(l_p, l_g) \ \wedge$$
$$\quad arg2_X(l_p) = retv_X(l_g)$$

Lemma 10.47 (BasicMapPut)**.** *In every sequential execution on a basic map, for every get method call g, if p is the latest preceding put method call with the same key argument then the return value of g is equal to the value argument of p. Formally,*

$$\forall m \in BasicMap \colon \forall X \in \mathbb{H}_B(m) \colon X \in Sequential \Rightarrow \tag{135}$$
$$\forall l_g, l_p \colon$$
$$\quad isXGet_{X,m}(l_g) \ \wedge$$
$$\quad isPutter_{X,m}(l_p, l_g) \ \wedge$$
$$\Rightarrow$$
$$\quad retv_X(l_g) = arg2_X(l_p)$$

10.3 Transactional Histories

Transactional Memory. The *transactional memory* is a singleton object *mem* that encapsulates a set of locations where each location, $i \in I$, $I = \{1, \ldots, m\}$ encapsulates a value v. The object *mem* has five methods $init_t()$, $read_t(i)$, $write_t(i, v)$, $commit_t()$ and $abort_t()$. The parameter t is the invoking transaction identifier. The method call $init_t()$ initializes t and returns ok. The method call $read_t(i)$ returns the value of location i or aborts t and returns $\mathbb{A}$. The method $write_t(i, v)$ writes v to location i and returns ok or aborts t and returns $\mathbb{A}$. The method $commit_t()$ tries to commit transaction t. If t is successfully committed, it returns $\mathbb{C}$; otherwise, it returns $\mathbb{A}$. The method $abort_t()$ aborts t and returns $\mathbb{A}$. The object *mem* can be implicit, that is $read_t(i)$ abbreviates $mem.read_T(i)$. The reserved values ok, $\mathbb{A}$, $\mathbb{C}$ denote successful completion of writes and, abortion and commitment of transactions respectively.

Transaction History. A *transaction history* H is an execution history such that $H|mem = H_{Init} \cdot H'$ with the following conditions. H_{Init} is the following history that initializes every location to v_0. $H_{Init} = l_{0i} \triangleright init_{T_0}() \cdot l_{00} \triangleright write_{T_0}(1, v_0){:}ok \cdot \ldots \cdot l_{0m} \triangleright write_{T_0}(m, v_0){:}ok \cdot l_{0c} \triangleright commit_{T_0}{:}\mathbb{C}$. For every $T \in H'$, the history $H'|T$ is a prefix of $E.E'$. The event sequence E is the initialization method call $l \triangleright init_T()$ (for some l), and then a sequence of reads $l \triangleright read_T(i){:}v$ and writes $l \triangleright write_T(i, v)$ (for some l, i, and v). The event sequence E' is one of the following sequences (for some l, i, and v): (1) $inv(l \triangleright read_T(i))$, $ret(l \triangleright \mathbb{A})$, (2) $inv(l \triangleright write_T(i, v))$, $ret(l \triangleright \mathbb{A})$, (3) $inv(l \triangleright commit_T())$, $ret(l \triangleright \mathbb{C})$, (4) $inv(l \triangleright commit_T())$, $ret(l \triangleright \mathbb{A})$, or (5) $inv(l \triangleright abort_T())$, $ret(l \triangleright \mathbb{A})$. Let $THistory$ denote the set of transaction histories. Let $Trans(H)$ denote the set of transactions of H. The projection of H on i, written $H|i$, denotes the subsequence of history H that contains exactly the events on location i. For a TM algorithm description π, let $\mathbb{H}(\pi)$ denote the set of complete transaction histories that result from execution of transactions with π.

11 Inference Rules

In this section, we now present the inference rules.

The judgements are of the form $\pi, \Gamma \vdash \mathcal{A}$ read assertion $\mathcal{A}$ is derived from the assumption assertions Γ for the specification π. The context Γ is defined as follows:

$$\Gamma \quad ::= \quad \cdot \mid \Gamma; \mathcal{A} \qquad \text{Context}$$

We present the classical first-order logic rules, the structure inference rules, the basic inference rules, and the synchronization object inference rules.

11.1 Classical First-order Logic Inference Rules

The classical inference rules are presented in Figure 16. The derived classical inference rules are presented in Figure 17.

The equivalence and arithmetic Rules are presented in Figure 18. The derived equivalence and arithmetic Rules are presented in Figure 19.

$$\textsc{Premise}\ \frac{}{\pi, \Gamma; \mathcal{A}; \Gamma' \vdash \mathcal{A}} \qquad \textsc{NegIntro}\ \frac{\pi, \Gamma; \mathcal{A} \vdash \mathcal{A}' \quad \pi, \Gamma; \mathcal{A} \vdash \neg\mathcal{A}'}{\pi, \Gamma \vdash \neg\mathcal{A}}$$

$$\textsc{ConjIntro}\ \frac{\pi, \Gamma \vdash \mathcal{A} \qquad \pi, \Gamma \vdash \mathcal{A}'}{\pi, \Gamma \vdash \mathcal{A} \ \wedge\ \mathcal{A}'} \qquad \textsc{ExcMid}\ \frac{}{\pi, \Gamma \vdash \mathcal{A} \ \vee\ \neg\mathcal{A}}$$

$$\textsc{ConjElimL}\ \frac{\pi, \Gamma \vdash \mathcal{A} \ \wedge\ \mathcal{A}'}{\pi, \Gamma \vdash \mathcal{A}} \qquad \textsc{ConjElimR}\ \frac{\pi, \Gamma \vdash \mathcal{A} \ \wedge\ \mathcal{A}'}{\pi, \Gamma \vdash \mathcal{A}'} \qquad \textsc{NegElim}\ \frac{\pi, \Gamma \vdash \mathcal{A} \quad \pi, \Gamma \vdash \neg\mathcal{A}}{\pi, \Gamma \vdash \mathcal{A}'}$$

$$\textsc{DisjIntroL}\ \frac{\pi, \Gamma \vdash \mathcal{A}}{\pi, \Gamma \vdash \mathcal{A} \ \vee\ \mathcal{A}'} \qquad \textsc{DisjIntroR}\ \frac{\pi, \Gamma \vdash \mathcal{A}'}{\pi, \Gamma \vdash \mathcal{A} \ \vee\ \mathcal{A}'} \qquad \textsc{UnivIntro}\ \frac{\pi, \Gamma \vdash \mathcal{A}[\ell := l] \quad l \notin \Gamma}{\pi, \Gamma \vdash \forall \ell : \mathcal{A}}$$

$$\textsc{DisjElim}\ \frac{\pi, \Gamma \vdash \mathcal{A} \ \vee\ \mathcal{A}' \quad \pi, \Gamma; \mathcal{A} \vdash \mathcal{A}'' \quad \pi, \Gamma; \mathcal{A}' \vdash \mathcal{A}''}{\pi, \Gamma \vdash \mathcal{A}''} \qquad \textsc{UnivElim}\ \frac{\pi, \Gamma \vdash \forall \ell : \mathcal{A}}{\pi, \Gamma \vdash \mathcal{A}[\ell := l]}$$

$$\textsc{CondIntro}\ \frac{\pi, \Gamma; \mathcal{A} \vdash \mathcal{A}'}{\pi, \Gamma \vdash \mathcal{A} \Rightarrow \mathcal{A}'} \qquad \textsc{ExistIntro}\ \frac{\pi, \Gamma \vdash \mathcal{A}[\ell := l]}{\pi, \Gamma \vdash \exists \ell : \mathcal{A}}$$

$$\textsc{CondElim}\ \frac{\pi, \Gamma \vdash \mathcal{A} \Rightarrow \mathcal{A}' \quad \pi, \Gamma \vdash \mathcal{A}}{\pi, \Gamma \vdash \mathcal{A}'} \qquad \textsc{ExistElim}\ \frac{\pi, \Gamma \vdash \exists \ell : \mathcal{A} \quad l \notin \Gamma \quad \pi, \Gamma; \mathcal{A}[\ell := l] \vdash \mathcal{A}'}{\pi, \Gamma \vdash \mathcal{A}'}$$

Figure 16. Classical Inference Rules

$$\textsc{DisjSyllL}\ \frac{\pi,\Gamma \vdash \mathcal{A} \vee \mathcal{A}' \quad \pi,\Gamma \vdash \neg\mathcal{A}}{\pi,\Gamma \vdash \mathcal{A}'} \qquad \textsc{DisjSyllR}\ \frac{\pi,\Gamma \vdash \mathcal{A} \vee \mathcal{A}' \quad \pi,\Gamma \vdash \neg\mathcal{A}'}{\pi,\Gamma \vdash \mathcal{A}} \qquad \textsc{CondElim'}\ \frac{\pi,\Gamma \vdash \mathcal{A} \Rightarrow \mathcal{A}' \quad \pi,\Gamma \vdash \neg\mathcal{A}'}{\pi,\Gamma \vdash \neg\mathcal{A}}$$

Figure 17. Derived Classical Inference Rules

$$\textsc{LRefl}\ \frac{}{\pi,\Gamma \vdash l = l} \qquad \textsc{ERefl}\ \frac{}{\pi,\Gamma \vdash e = e} \qquad \textsc{Zero}\ \frac{}{\pi,\Gamma \vdash \neg(1 = 0)}$$

$$\textsc{LSubs}\ \frac{\pi,\Gamma \vdash l = l' \quad \pi,\Gamma \vdash \mathcal{A}}{\pi,\Gamma \vdash \mathcal{A}[l := l']} \qquad \textsc{LSubs}\ \frac{\pi,\Gamma \vdash e = e' \quad \pi,\Gamma \vdash \mathcal{A}}{\pi,\Gamma \vdash \mathcal{A}[e := e']}$$

Figure 18. Equivalence and Arithmetic Rules

$$\textsc{LSym}\ \frac{\pi,\Gamma \vdash l = l'}{\pi,\Gamma \vdash l' = l} \qquad \textsc{LTrans}\ \frac{\pi,\Gamma \vdash l = l' \quad \pi,\Gamma \vdash l' = l''}{\pi,\Gamma \vdash l = l''} \qquad \textsc{ESym}\ \frac{\pi,\Gamma \vdash e = e'}{\pi,\Gamma \vdash e' = e} \qquad \textsc{ETrans}\ \frac{\pi,\Gamma \vdash e = e' \quad \pi,\Gamma \vdash e' = e''}{\pi,\Gamma \vdash e = e''}$$

Figure 19. Derived Equivalence and Arithmetic Rules

11.2 Structure Inference Rules

The structure inference rules that axiomatize the relation of the program structure and the execution. The structure inference rules are presented in Figures 20 The derived structure inference rules are presented in Figure 21. The derived inference rules can be derived from the basic rules. Please see Section 15.3 for notes on the derivation of the derived rules.

The rule ID states that components of method calls in the history originate from components of method calls in the program. The object, arguments and other components of an executed method call labeled ς'c can be derived from prefixing the object, arguments and other components of the method call annotated with c in the program with the pre-label ς. Note that the pre-label ς is a constant c' when the method call c is executed inside the body of a *this* method call annotated with c'. The pre-label ς is ϵ when c is the annotation of a *this* method call.

The rule SRC states that every executed method originates from a call site in the program. If a method n on an object with the base name ϕ is executed, it is from one of the call sites where n is called on ϕ in the program.

The rule OCONTROL states when a *this* method call is executed. A *this* method call is executed if an only if its execution condition is satisfied.

The rule ICONTROL states when a method call in the body of a *this* method call is executed. A method call (annotated with) c' in a *this* method call (annotated with) c is executed if and only if c is executed, the execution condition of c' is satisfied and no return statement before c' is executed.

The rule P2X states that the program order is preserved in the execution order. If a method call annotated with c_1 is ordered before a method call annotated with c_2 in the program, and methods labeled ς'c_1 and ς'c_2 are executed, then ς'c_1 is executed before ς'c_2.

The rule OX2IX states that the execution order of two *this* method calls implies the execution order of method calls in their bodies. If a *this* method call c_1 is executed before another *this* method call c_2, then every executed method call of the body of c_1 is executed before every executed method call of the body of c_2.

The rule TSEQ states that every thread is sequential. Every two *this* method calls by the same thread are ordered in the execution order. Similarly, every two method calls on base objects by the same thread are ordered in the execution order.

The rule CALLER states that if a *this* method call is executed, its parameters and arguments are equal and that one of the return statements in its body is executed and its return value is equal to the value that the executed return statement returns.

The rule CALLEE states that if a method call in the body of a *this* method call is executed, then the *this* method call is executed and the parameters and the arguments of the *this* method call are equal.

The rule RET states that if a return statement of the body of a *this* method call is executed, then the *this* method call is executed and the parameters and the arguments of the *this* method call are equal and the return value of the *this* method call is the value that the return statement returns.

The rule TLOCAL states that every two executed method calls on the same thread-local object are from the same thread.

The rule TREAL states that if a thread is ordered before another thread, then every method call from the former is executed before every method call from the latter.

The rule IX2OX states that if two method calls in the body of two *this* method calls execute in order by the same thread, then the two *this* method calls execute in the same order.

ID

$$\frac{obj_\pi(c) = \theta \qquad name_\pi(c) = n \qquad thread_\pi(c) = \tau \qquad arg_\pi(c) = u \qquad retv_\pi(c) = x \qquad \pi,\Gamma \vdash exec(\varsigma\text{'}c)}{\pi,\Gamma \vdash obj(\varsigma\text{'}c) = \varsigma\text{'}\theta \wedge name(\varsigma\text{'}c) = n \wedge thread(\varsigma\text{'}c) = \varsigma\text{'}\tau \wedge arg(\varsigma\text{'}c) = \varsigma\text{'}u \wedge retv(\varsigma\text{'}c) = \varsigma\text{'}x}$$

SRC

$$\frac{\pi,\Gamma \vdash exec(\varsigma\text{'}c) \qquad \pi,\Gamma \vdash obj(\varsigma\text{'}c) = \theta \qquad \pi,\Gamma \vdash name(\varsigma\text{'}c) = n \qquad Calls_\pi(basename(\theta), n) = \{\overline{c_i}\}}{\pi,\Gamma \vdash \bigvee_{i=1..n} c = c_i}$$

OCONTROL

$$\frac{c \in Labels(\mathcal{P})}{\pi,\Gamma \vdash exec(c) \Leftrightarrow cond_\pi(c)}$$

ICONTROL

$$\frac{Labels(name_\pi(c)) = \{\overline{c_i}\} \qquad PreReturns_\pi(c') = \{\overline{c_r}\}}{\pi,\Gamma \vdash exec(c\text{'}c') \Leftrightarrow (exec(c) \wedge \bigvee_{c_i} c' = c_i \wedge c\text{'}cond_\pi(c') \wedge \bigwedge_{c_r} \neg exec(c\text{'}c_r))}$$

P2X

$$\frac{c_1 \rightarrow_\pi c_2 \qquad \pi,\Gamma \vdash exec(\varsigma\text{'}c_1) \qquad \pi,\Gamma \vdash exec(\varsigma\text{'}c_2)}{\pi,\Gamma \vdash \varsigma\text{'}c_1 \prec \varsigma\text{'}c_2}$$

OX2IX

$$\frac{\pi,\Gamma \vdash c_1 \prec c_2 \qquad \pi,\Gamma \vdash exec(c_1\text{'}c_3) \qquad \pi,\Gamma \vdash exec(c_2\text{'}c_4)}{\pi,\Gamma \vdash c_1\text{'}c_3 \prec c_2\text{'}c_4}$$

TSEQ

$$\frac{\pi,\Gamma \vdash exec(l_1) \qquad \pi,\Gamma \vdash exec(l_2) \qquad \pi,\Gamma \vdash thread(l_1) = thread(l_2) \qquad \pi,\Gamma \vdash obj(l_1) = obj(l_2) = \textbf{this} \vee (\neg obj(l_1) = \textbf{this} \wedge \neg obj(l_2) = \textbf{this})}{\pi,\Gamma \vdash l_1 \prec l_2 \vee l_2 \prec l_1 \vee l_1 = l_2}$$

CALLER

$$\frac{tpar_\pi(n) = t \qquad par1_\pi(n) = x \qquad Returns_\pi(n) = \{\overline{c_i}\} \qquad \pi,\Gamma \vdash exec(c) \qquad \pi,\Gamma \vdash obj(c) = \textbf{this} \wedge name(c) = n}{\pi,\Gamma \vdash c\text{'}t = thread(c) \wedge c\text{'}x = arg(c) \wedge \bigvee_{i=1..n} exec(c\text{'}c_i) \wedge arg1(c\text{'}c_i) = retv(c)}$$

CALLEE

$$\frac{tpar_\pi(n) = t \qquad par1_\pi(n) = x \qquad c' \in Labels_\pi(n) \qquad \pi,\Gamma \vdash exec(\varsigma\text{'}c')}{\pi,\Gamma \vdash \neg(\varsigma = \epsilon) \wedge exec(\varsigma) \wedge obj(\varsigma) = \textbf{this} \wedge name(\varsigma) = n \wedge thread(\varsigma) = \varsigma\text{'}t \wedge arg(\varsigma) = \varsigma\text{'}x}$$

RET

$$\frac{tpar_\pi(n) = t \qquad par1_\pi(n) = x \qquad c' \in Returns_\pi(n) \qquad \pi,\Gamma \vdash exec(c\text{'}c')}{\pi,\Gamma \vdash exec(c) \wedge obj(c) = \textbf{this} \wedge name(c) = n \wedge thread(c) = c\text{'}t \wedge arg(c) = c\text{'}x \wedge retv(c) = arg1(c\text{'}c')}$$

TLOCAL

$$\frac{\mathcal{T}(basename(o)) = ThreadLocal\ st \qquad \pi,\Gamma \vdash exec(l_1) \wedge exec(l_2) \qquad \pi,\Gamma \vdash obj(l_1) = obj(l_2) = o}{\pi,\Gamma \vdash thread(l_1) = thread(l_2)}$$

TREAL

$$\frac{\pi,\Gamma \vdash \tau \lll \tau' \qquad \pi,\Gamma \vdash exec(l) \wedge thread(l) = \tau \qquad \pi,\Gamma \vdash exec(l') \wedge thread(l') = \tau'}{\pi,\Gamma \vdash l \prec l'}$$

Figure 20. Structure Inference Rules. All of the rules have the side condition $\pi = (\mathcal{T}, \mathcal{D}, \mathcal{P})$

IX2OX

$$\frac{\pi,\Gamma \vdash c_1\text{'}c_3 \prec c_2\text{'}c_4 \qquad \pi,\Gamma \vdash thread(c_1\text{'}c_3) = thread(c_2\text{'}c_4)}{\pi,\Gamma \vdash c_1 \prec c_2 \vee c_1 = c_2}$$

Figure 21. Derived Structure Inference Rules

XASym
$$\frac{\pi,\Gamma \vdash l \prec l'}{\pi,\Gamma \vdash \neg(l' \prec l) \ \wedge \ \neg(l' \sim l) \ \wedge \ \neg(l' = l)}$$

XTrans
$$\frac{\pi,\Gamma \vdash l \prec l' \qquad \pi,\Gamma \vdash l' \prec l''}{\pi,\Gamma \vdash l \prec l''}$$

XXTrans
$$\frac{\pi,\Gamma \vdash l_1 \prec l_2 \qquad \pi,\Gamma \vdash l_3 \prec l_4 \qquad \pi,\Gamma \vdash l_2 \sim l_3}{\pi,\Gamma \vdash l_1 \prec l_4}$$

XTotal
$$\frac{\pi,\Gamma \vdash exec(l) \ \wedge \ exec(l')}{\pi,\Gamma \vdash (l \prec l') \ \vee \ (l' \prec l) \ \vee \ (l \sim l') \ \vee \ (l = l')}$$

X2L
$$\frac{\mathcal{T}_{base}(o) \in LT \qquad \pi,\Gamma \vdash obj(l) = obj(l') = o \qquad \pi,\Gamma \vdash l \prec l'}{\pi,\Gamma \vdash l \prec_o l'}$$

LASym
$$\frac{\pi,\Gamma \vdash l \prec_o l'}{\pi,\Gamma \vdash \neg(l' \prec_o l) \ \wedge \ \neg(l = l')}$$

LTrans
$$\frac{\pi,\Gamma \vdash l \prec_o l' \qquad \pi,\Gamma \vdash l' \prec_o l''}{\pi,\Gamma \vdash l \prec_o l''}$$

LTotal
$$\frac{\mathcal{T}_{base}(o) \in LT \cup SCT \qquad \pi,\Gamma \vdash exec(l) \ \wedge \ exec(l') \qquad \pi,\Gamma \vdash obj(l) = obj(l') = o}{\pi,\Gamma \vdash (l \prec_o l') \ \vee \ (l' \prec_o l) \ \vee \ (l = l')}$$

All of the rules have the side condition $\pi = (\mathcal{T}, \mathcal{D}, \mathcal{P})$

Figure 22. Basic Inference Rules

P2L
$$\frac{c_1 \rightarrow_\pi c_2 \qquad \pi,\Gamma \vdash exec(\varsigma\text{'}c_1) \qquad \pi,\Gamma \vdash exec(\varsigma\text{'}c_2) \qquad \mathcal{T}_{base}(o) \in LT \qquad \pi,\Gamma \vdash obj(\varsigma\text{'}c_1) = obj(\varsigma\text{'}c_2) = o}{\pi,\Gamma \vdash \varsigma\text{'}c_1 \prec_o \varsigma\text{'}c_2}$$

X2X
$$\frac{\pi,\Gamma \vdash l \prec l'}{\pi,\Gamma \vdash exec(l) \ \wedge \ exec(l')}$$

L2X
$$\frac{\mathcal{T}_{base}(o) \in LT \cup SCT \qquad \pi,\Gamma \vdash l \prec_o l'}{\pi,\Gamma \vdash exec(l) \ \wedge \ exec(l') \ \wedge \ obj(l) = obj(l') = o}$$

XLTrans
$$\frac{\mathcal{T}_{base}(o) \in LT \qquad \pi,\Gamma \vdash l_1 \prec l_2 \qquad \pi,\Gamma \vdash l_3 \prec l_4 \qquad \pi,\Gamma \vdash l_2 \prec_o l_3}{\pi,\Gamma \vdash l_1 \prec l_4}$$

X2L'
$$\frac{\mathcal{T}_{base}(o) \in LT \qquad \pi,\Gamma \vdash l \prec_o l'}{\pi,\Gamma \vdash l \precsim l'}$$

Figure 23. Derived Basic Inference Rules

11.3 Basic Inference Rules

The basic inference rules axiomatize the properties of the execution and linearization orders and their interdependence. The basic inference rules state are presented in 22. The derived basic inference rules are presented in Figure 23. We explain each rule in turn.

The rule XASym states the asymmetry property of the execution order. If a method call is executed before another method call, then the latter is not executed before the former and they are not executed concurrently.

The rule XTrans states the transitivity property of the precedence execution order. The rule XXTrans states the transitivity of the sequence of precedence, concurrency and precedence execution relations. If l_1 is executed before l_2, l_2 is executed (before or) concurrent to l_3 and l_3 is executed before l_4, then l_1 is executed before l_4.

The rule XTotal states the totality property of the precedence and concurrency execution relations. Every two method calls either execute in order or concurrently.

The rule X2X states that if a method call is executed before another one, then obviously both are executed.

The rule X2L states the real-time-preservation property of linearization orders. The execution order of two method calls on a linearizable object is preserved in the linearization order.

The rule LASym states the asymmetry property of linearization orders. If a method call is linearized before another one, then the latter is not linearized before the former.

The rule LTrans states the transitivity property of linearization orders.

The rule LTotal states the totality property of linearization orders.

The rule L2X states that if a method call is linearized before another one, then obviously both are executed.

The rule P2L states that the program order of two method calls on a linearizable object is preserved in the linearization order.

The rule XLTrans is a form of "transitivity" rule for judgements about the execution order $\prec$ and the linearization order $\prec_o$ for a linearizable object o. If l_1 is executed before l_2, l_2 is linearized before l_3 and l_3 is executed before l_4, then l_1 is executed before l_4.

The rule X2L' states the contra-positive of the rule X2L.

11.4 Synchronization Object Inference Rules

The synchronization object inference rules axiomatize the properties of common synchronization object types. We consider each type in turn.

Basic and Atomic Register. The basic and atomic register inference rules are presented in Figure 24.

The rule AReg states that for every read method call l_R on an atomic register, there is a write method call ℓ_W on it that writes the same value that l_R returns and ℓ_W is the last write method call that is linearized before l_R.

A method call ℓ is race-free $isRaceFree_r(\ell)$ if an only if there is no write method call that executes concurrent to it. A register reg is sequentially-written $isSequentiallyWritten(reg)$ if and only if every pair of write method calls on it are ordered in the execution order or in other words, every write method call on it is race-free.

$$\textsc{AReg}\ \frac{\mathcal{T}_{base}(reg) = AtomicRegister \quad \pi,\Gamma \vdash isRead_{reg}(l_R)}{\pi,\Gamma \vdash \exists \ell_W : isWriter_{reg}(\ell_W, l_R) \wedge retv(l_R) = arg1(\ell_W)}$$

$$\textsc{BReg}\ \frac{\mathcal{T}_{base}(reg) = BasicRegister \quad \pi,\Gamma \vdash isSequentiallyWritten(reg) \quad \pi,\Gamma \vdash isRead_{reg}(l_R) \quad \pi,\Gamma \vdash isRaceFree_{reg}(l_R)}{\pi,\Gamma \vdash \exists \ell_W : isEWriter_{reg}(\ell_W, l_R) \wedge retv(l_R) = arg1(\ell_W)}$$

$$\begin{aligned}
&isRead_r(\ell_R) \Leftrightarrow \\
&\quad exec(\ell_R) \ \wedge\ obj(\ell_R) = r \ \wedge\ name(\ell_R) = read \\
&isWrite_r(\ell_W) \Leftrightarrow \\
&\quad exec(\ell_W) \ \wedge\ obj(\ell_W) = r \ \wedge\ name(\ell_W) = write \\
&isWriter_r(\ell_W, \ell_R) \Leftrightarrow \\
&\quad isWrite_r(\ell_W) \ \wedge\ \ell_W \prec_r \ell_R \wedge \\
&\quad \forall \ell'_W : isWrite_r(\ell'_W) \Rightarrow (\ell'_W \preceq_r \ell_W \ \vee\ \ell_R \prec_r \ell'_W) \\
&isEWriter_r(\ell_W, \ell_R) \Leftrightarrow \\
&\quad isWrite_r(\ell_W) \ \wedge\ \ell_W \prec \ell_R \wedge \\
&\quad \forall \ell'_W : isWrite_r(\ell'_W) \Rightarrow (\ell'_W \preceq \ell_W \ \vee\ \ell_R \prec \ell'_W) \\
&isSequential(o) \Leftrightarrow \\
&\quad \forall \ell, \ell' : (exec(\ell) \ \wedge\ exec(\ell') \ \wedge\ obj(\ell) = o \ \wedge\ obj(\ell') = o) \Rightarrow \\
&\qquad (\ell \preceq \ell' \ \vee\ \ell' \prec \ell) \\
&isRaceFree_r(\ell) \Leftrightarrow \\
&\quad \forall \ell_W : isWrite_r(\ell_W) \Rightarrow (\ell_W \prec \ell \ \vee\ \ell \prec \ell_W) \\
&isSequentiallyWritten(r) \Leftrightarrow \\
&\quad \forall \ell_w : isWrite_r(\ell_w) \Rightarrow isRaceFree_r(\ell_w)
\end{aligned}$$

Figure 24. Register Inference Rules.

$$\textsc{AReg'}\ \frac{\mathcal{T}_{base}(reg) = AtomicRegister \quad \pi,\Gamma \vdash isRead_{reg}(l_R) \quad \pi,\Gamma \vdash isWriter_{reg}(l_W, l_R)}{\pi,\Gamma \vdash arg1(l_W) = retv(l_R)}$$

$$\textsc{BReg'}\ \frac{\mathcal{T}_{base}(reg) = BasicRegister \quad \pi,\Gamma \vdash isRead_{reg}(l_R) \quad \pi,\Gamma \vdash isSequential(reg)}{\pi,\Gamma \vdash \exists \ell_W : isEWriter_{reg}(\ell_W, l_R) \wedge retv(l_R) = arg1(\ell_W)}$$

$$\textsc{TReg}\ \frac{\mathcal{T}(reg) = ThreadLocal\ BasicRegister \quad \pi,\Gamma \vdash isRead_{reg[\tau]}(l_R)}{\pi,\Gamma \vdash \exists \ell_W : isEWriter_{reg[\tau]}(\ell_W, l_R) \wedge retv(l_R) = arg1(\ell_W) \wedge thread(\ell_W) = \tau}$$

Figure 25. Derived Register Inference Rules

The rule BReg states that if a basic register *reg* is sequentially-written, for every race-free read method call l_R on *reg*, there is a write method call ℓ_W on *reg* that writes the same value that l_R returns and ℓ_W is the last write method call that is executed before l_R. Note that this models Lamport's notion of safe registers [26].

The derived register inference rules are presented in Figure 25.

The rule AReg' states that for every read method call l_R on an atomic register, if l_W is the last write method call that is linearized before l_R, then l_W writes the same value that l_R returns.

An object *o* is accessed sequentially $isSequential(o)$ if and only if every pair of method calls on it are ordered in the execution order.

The rule BReg' states that if a basic register *reg* is accessed sequentially, for every read method call l_R on *reg*, there is a write method call ℓ_W on *reg* that writes the same value that l_R returns and ℓ_W is the last write method call that is executed before l_R.

The rule TReg states that for every read method call l_R on a thread-local register *reg*, there is a write method call ℓ_W on *reg* that writes the same value that l_R returns and ℓ_W is the last write method call that is executed before l_R.

$$\textsc{CASRegRead}\quad \frac{\mathcal{T}_{base}(reg) = AtomicCASRegister \qquad \pi,\Gamma \vdash isRead_{reg}(l_R)}{\pi,\Gamma \vdash \exists \ell_W : isCWriter_{reg}(\ell_W, l_R) \wedge retv(l_R) = writtenValue(\ell_W)}$$

$$\textsc{CASRegCAST}\quad \frac{\mathcal{T}_{base}(reg) = AtomicCASRegister \qquad \pi,\Gamma \vdash isCAS_{reg}(l_C) \qquad \pi,\Gamma \vdash isCWriter_{reg}(l_W, l_R) \qquad \pi,\Gamma \vdash arg1(l_C) = writtenValue(l_W)}{\pi,\Gamma \vdash retv(l_C) = true}$$

$$\textsc{CASRegCASF}\quad \frac{\mathcal{T}_{base}(reg) = AtomicCASRegister \qquad \pi,\Gamma \vdash isCAS_{reg}(l_C) \qquad \pi,\Gamma \vdash isCWriter_{reg}(l_W, l_R) \qquad \pi,\Gamma \vdash \neg(arg1(l_C) = writtenValue(l_W))}{\pi,\Gamma \vdash retv(l_C) = false}$$

$$\begin{aligned}
&isRead_r(\ell_R) \Leftrightarrow \\
&\quad exec(\ell_R) \ \wedge \ obj(\ell_R) = r \ \wedge \ name(\ell_R) = read \\
&isWrite_r(\ell_R) \Leftrightarrow \\
&\quad exec(\ell_R) \ \wedge \ obj(\ell_R) = r \ \wedge \ name(\ell_R) = write \\
&isCAS_r(\ell_R) \Leftrightarrow \\
&\quad exec(\ell_R) \ \wedge \ obj(\ell_R) = r \ \wedge \ name(\ell_R) = cas \\
&isCWrite_r(\ell_W) \Leftrightarrow \\
&\quad isWrite_r(\ell_W) \ \vee \ (isCAS_r(\ell_W) \ \wedge \ retv(\ell_W) = true) \\
&isCWriter_r(\ell_W, \ell_R) \Leftrightarrow \\
&\quad isCWrite_r(\ell_W) \ \wedge \ \ell_W \prec_r \ell_R \wedge \\
&\quad \forall \ell'_W : isCWrite_r(\ell'_W) \Rightarrow (\ell'_W \preceq_r \ell_W \ \vee \ \ell_R \prec_r \ell'_W) \\
&writtenValue(\ell) = \\
&\quad \begin{cases} arg1(\ell) & \text{if } obj(\ell) = write \\ arg2(\ell) & \text{if } obj(\ell) = cas \end{cases}
\end{aligned}$$

Figure 26. CAS Register Inference Rules.

$$\textsc{CASRegRead'}\quad \frac{\mathcal{T}_{base}(reg) = CASAtomicRegister \qquad \pi,\Gamma \vdash isRead_{reg}(l_R) \qquad \pi,\Gamma \vdash isCWriter_{reg}(l_W, l_R)}{\pi,\Gamma \vdash retv(l_R) = writtenValue(l_W)}$$

Figure 27. Derived CAS Register Inference Rules

CAS Atomic Register. The cas register inference rules are presented in Figure 26.

A method call ℓ_W on an atomic cas register r is a successful write $isCWrite_r(\ell_W)$, if and only if it is a write method call or a successful cas method call. The written value $writtenValue(\ell)$ of a successful write method call ℓ is its first argument if it is a write method call and its second argument if it a successful cas method call.

The rule CASRegRead states that for every read method call l_R on an atomic cas register, there is a successful write ℓ_W that writes the same value that l_R has returned and ℓ_W is the last successful write that is linearized before l_R.

The rule CASRegCAST and the rule CASRegCASF state that a cas method call l_C on an atomic cas register returns *true* if the written value of the last successful write linearized before l_C is equal to the first argument of l_C, and returns $false$ otherwise.

The derived cas register inference rules are presented in Figure 27.

The rule CASRegRead' states that for every read method call l_R on an atomic cas register, the last successful write that is linearized before l_R writes the same value that l_R returns.

Lock and Try-Lock. The preliminary definitions are presented in Figure 28 and the lock and try-lock inference rules are presented in Figure 29.

Ownership for a lock l is respected, $isOwnerRespecting(l)$ if and only if every thread unlocks l only if it has already locked l and has not unlocked it since then.

The rule Lock states that if ownership is respected for a lock l and a *lock* method call on l (by a thread t_1) is linearized before an *unlock* method call on l (by a thread t_2), then an *unlock* method call on l by t_1 is linearized before a *lock* method call on l by t_2.

The rule LockReadL states that if ownership is respected for a lock l and an *unlock* method call on l (by a thread t) is linearized after a *read* method call on l that returns $false$, then a *lock* method call on l by t is linearized after the *read* method call.

The rule LockReadR states that if ownership is respected for a lock l and a *lock* method call on l (by a thread t) is linearized before a *read* method call on l that returns $false$, then an *unlock* method call on l by t is linearized before the *read* method call.

The rule LockReadM states that if ownership is respected for a lock l and a *read* method call on l (by a thread t) is linearized between a pair of matching *lock* and *unlock* method call on l, then the read method call returns *true*.

There are four similar rules for try-locks. Instead of *lock* method calls, these rules concern successful lock method calls that are *lock* and successful *tryLock* method calls.

$$
\begin{aligned}
&isLock_o(\ell) \Leftrightarrow \\
&\quad exec(\ell) \ \wedge \ obj(\ell) = o \ \wedge \ name(\ell) = lock \\
&isUnlock_o(\ell) \Leftrightarrow \\
&\quad exec(\ell) \ \wedge \ obj(\ell) = o \ \wedge \ name(\ell) = unlock \\
&isRead_o(\ell) \Leftrightarrow \\
&\quad exec(\ell) \ \wedge \ obj(\ell) = o \ \wedge \ name(\ell) = read \\
&isTryLock_o(\ell) \Leftrightarrow \\
&\quad exec(\ell) \ \wedge \ obj(\ell) = o \ \wedge \ name(\ell) = tryLock \\
&isTLock_o(\ell) \Leftrightarrow \\
&\quad isLock_o(\ell) \ \vee \ (isTryLock_o(\ell) \ \wedge \ retv(\ell) = true) \\
&noUnlockBetween_o(\ell_l, \ell_u) \Leftrightarrow \\
&\quad \forall \ell'_u : \\
&\qquad (isXUnlock_{X,o}(\ell'_u) \wedge \\
&\qquad thread_X(\ell_l) = thread_X(\ell'_u)) \Rightarrow \\
&\qquad (\ell'_u \prec \ell_l \ \vee \ \ell_u \preceq \ell'_u)
\end{aligned}
$$

$$
\begin{aligned}
&isOwnerRespecting(o) \Leftrightarrow \\
&\quad \forall \ell : isUnlock_o(\ell) \Rightarrow \\
&\quad \exists \ell' : isTLock_o(\ell') \wedge \\
&\qquad thread(\ell') = thread(\ell) \wedge \\
&\qquad \ell' \prec \ell \wedge \\
&\qquad \forall \ell'' : \\
&\qquad\quad (isUnLock_o(\ell'') \wedge \\
&\qquad\quad thread(\ell'') = thread(\ell)) \\
&\qquad \Rightarrow \\
&\qquad\quad \ell'' \prec \ell' \ \vee \ \ell \preceq \ell''
\end{aligned}
$$

Figure 28. Preliminary definitions for Lock and TryLock Inference Rules.

Lock

$$\frac{\begin{array}{c}\mathcal{T}_{base}(o) = Lock \\ \pi,\Gamma \vdash isOwnerRespecting(o) \\ \pi,\Gamma \vdash isLock_o(l_{l_1}) \qquad \pi,\Gamma \vdash isUnlock_o(l_{u_2}) \\ \pi,\Gamma \vdash l_{l_1} \prec_o l_{u_2}\end{array}}{\begin{array}{c}\pi,\Gamma \vdash \exists \ell_{u_1}, \ell_{l_2}: \\ isUnlock_o(\ell_{u_1}) \ \wedge \ thread(\ell_{u_1}) = thread(l_{l_1}) \wedge \\ isLock_o(\ell_{l_2}) \ \wedge \ thread(\ell_{l_2}) = thread(l_{u_2}) \wedge \\ \ell_{u_1} \prec_o \ell_{l_2}\end{array}}$$

TryLock

$$\frac{\begin{array}{c}\mathcal{T}_{base}(o) = TryLock \\ \pi,\Gamma \vdash isOwnerRespecting(o) \\ \pi,\Gamma \vdash isTLock_o(l_{l_1}) \qquad \pi,\Gamma \vdash isUnlock_o(l_{u_2}) \\ \pi,\Gamma \vdash l_{l_1} \prec_o l_{u_2}\end{array}}{\begin{array}{c}\pi,\Gamma \vdash \exists \ell_{u_1}, \ell_{l_2}: \\ isUnlock_o(\ell_{u_1}) \ \wedge \ thread(\ell_{u_1}) = thread(l_{l_1}) \wedge \\ isTLock_o(\ell_{l_2}) \ \wedge \ thread(\ell_{l_2}) = thread(l_{u_2}) \wedge \\ \ell_{u_1} \prec_o \ell_{l_2}\end{array}}$$

LockReadL

$$\frac{\begin{array}{c}\mathcal{T}_{base}(o) = Lock \\ \pi,\Gamma \vdash isOwnerRespecting(o) \\ \pi,\Gamma \vdash isUnlock_o(l_{u_1}) \\ \pi,\Gamma \vdash isRead(l_{r_2}) \ \wedge \ retv(l_{r_2}) = false \\ \pi,\Gamma \vdash l_{r_2} \prec_o l_{u_1}\end{array}}{\begin{array}{c}\pi,\Gamma \vdash \exists \ell_{l_1}: \\ isLock_o(\ell_{l_1}) \ \wedge \ thread(\ell_{l_1}) = thread(l_{u_1}) \wedge \\ l_{r_2} \prec_o \ell_{l_1}\end{array}}$$

TryLockReadL

$$\frac{\begin{array}{c}\mathcal{T}_{base}(o) = TryLock \\ \pi,\Gamma \vdash isOwnerRespecting(o) \\ \pi,\Gamma \vdash isUnlock_o(l_{u_1}) \\ \pi,\Gamma \vdash isRead(l_{r_2}) \ \wedge \ retv(l_{r_2}) = false \\ \pi,\Gamma \vdash l_{r_2} \prec_o l_{u_1}\end{array}}{\begin{array}{c}\pi,\Gamma \vdash \exists \ell_{l_1}: \\ isTLock_o(\ell_{l_1}) \ \wedge \ thread(\ell_{l_1}) = thread(l_{u_1}) \wedge \\ l_{r_2} \prec_o \ell_{l_1}\end{array}}$$

LockReadR

$$\frac{\begin{array}{c}\mathcal{T}_{base}(o) = Lock \\ \pi,\Gamma \vdash isOwnerRespecting(o) \\ \pi,\Gamma \vdash isLock_o(l_{l_1}) \\ \pi,\Gamma \vdash isRead(l_{r_2}) \ \wedge \ retv(l_{r_2}) = false \\ \pi,\Gamma \vdash l_{l_1} \prec_o l_{r_2}\end{array}}{\begin{array}{c}\pi,\Gamma \vdash \exists \ell_{u_1}: \\ isUnlock_o(\ell_{u_1}) \ \wedge \ thread(\ell_{u_1}) = thread(l_{l_1}) \wedge \\ \ell_{u_1} \prec_o l_{r_2}\end{array}}$$

TryLockReadR

$$\frac{\begin{array}{c}\mathcal{T}_{base}(o) = TryLock \\ \pi,\Gamma \vdash isOwnerRespecting(o) \\ \pi,\Gamma \vdash isTLock_o(l_{l_1}) \\ \pi,\Gamma \vdash isRead(l_{r_2}) \ \wedge \ retv(l_{r_2}) = false \\ \pi,\Gamma \vdash l_{l_1} \prec_o l_{r_2}\end{array}}{\begin{array}{c}\pi,\Gamma \vdash \exists \ell_{u_1}: \\ isUnlock_o(\ell_{u_1}) \ \wedge \ thread(\ell_{u_1}) = thread(l_{l_1}) \wedge \\ \ell_{u_1} \prec_o l_{r_2}\end{array}}$$

LockReadM

$$\frac{\begin{array}{c}\mathcal{T}_{base}(o) = Lock \\ \pi,\Gamma \vdash isOwnerRespecting(o) \\ \pi,\Gamma \vdash isLock_o(l_{l_1}) \qquad \pi,\Gamma \vdash isUnlock_o(l_{u_1}) \\ \pi,\Gamma \vdash thread(l_{u_1}) = thread(l_{l_1}) \\ \pi,\Gamma \vdash noUnlockBetween_o(l_{l_1}, l_{u_1}) \\ \pi,\Gamma \vdash isRead(l_{r_2}) \\ \pi,\Gamma \vdash l_{l_1} \prec_o l_{r_2} \qquad \pi,\Gamma \vdash l_{r_2} \prec_o l_{u_1}\end{array}}{\pi,\Gamma \vdash retv(l_{r_2}) = true}$$

TryLockReadM

$$\frac{\begin{array}{c}\mathcal{T}_{base}(o) = TryLock \\ \pi,\Gamma \vdash isOwnerRespecting(o) \\ \pi,\Gamma \vdash isTLock_o(l_{l_1}) \qquad \pi,\Gamma \vdash isUnlock_o(l_{u_1}) \\ \pi,\Gamma \vdash thread(l_{u_1}) = thread(l_{l_1}) \\ \pi,\Gamma \vdash noUnlockBetween_o(l_{l_1}, l_{u_1}) \\ \pi,\Gamma \vdash isRead(l_{r_2}) \\ \pi,\Gamma \vdash l_{l_1} \prec_o l_{r_2} \qquad \pi,\Gamma \vdash l_{r_2} \prec_o l_{u_1}\end{array}}{\pi,\Gamma \vdash retv(l_{r_2}) = true}$$

Figure 29. Lock and TryLock Inference Rules.

Strong Counter. The strong counter inference rules are presented in Figures 30 and 31.

The rule SCounter states that the return value of every method call that is linearized before an *iaf* method call is smaller than the return value of the *iaf* method call.

The rule SCounter' states that if the return value of a method call is greater than the return value of an *iaf* method call, then it is linearized after the *iaf* method call.

$$\textsc{SCounter}\quad \frac{\begin{array}{c}\mathcal{T}_{base}(o) = SCounter \\ \pi,\Gamma \vdash obj(l_1) = o \\ \pi,\Gamma \vdash obj(l_2) = o \ \wedge \ name(l_2) = iaf \\ \pi,\Gamma \vdash l_1 \prec_o l_2\end{array}}{\pi,\Gamma \vdash retv(l_1) < retv(l_2)}$$

Figure 30. SCounter Rules

$$\textsc{SCounter'}\quad \frac{\begin{array}{c}\mathcal{T}_{base}(o) = SCounter \\ \pi,\Gamma \vdash exec(l_1) \ \wedge \ obj(l_1) = o \\ \pi,\Gamma \vdash exec(l_2) \ \wedge \ obj(l_2) = o \ \wedge \ name(l_2) = iaf \\ \pi,\Gamma \vdash retv(l_1) > retv(l_2)\end{array}}{\pi,\Gamma \vdash l_2 \prec_o l_1}$$

Figure 31. Derived SCounter Rules

Basic Set and Basic Map. The Set and Map inference rules are presented in Figure 32.

An object o is accessed sequentially $isSequential(o)$ if and only if every pair of method calls on it are ordered in the execution order.

The rule BasicSetContains states that if a basic set s is accessed sequentially, for every *contains* method call on s that returns *true*, there is a preceding *add* method call on s with the same argument.

The rule BasicSetAdd states that if a basic set s is accessed sequentially, every *contains* method call on s that succeeds an *add* method call on s with the same argument returns *true*.

The rule BasicMapGet states that if a basic map m is accessed sequentially, for every *get* method call l_g on m that does not return $\perp$, there exists a *put* method call ℓ_p on m with the same key argument such that the value argument of p is equal to the return value of l_g and ℓ_p is the latest preceding *put* method call on m with the same key argument.

BasicSetContains

$$\frac{\mathcal{T}_{base}(s) = BasicSet \quad \pi,\Gamma \vdash isSequential(s) \quad \pi,\Gamma \vdash isContains_s(l_c) \ \wedge \ retv(l_c) = true}{\pi,\Gamma \vdash \exists \ell_a : isAdd_s(\ell_a) \wedge arg1(\ell_a) = arg1(l_c) \ \wedge \ \ell_a \prec l_c}$$

BasicMapGet

$$\frac{\mathcal{T}_{base}(m) = BasicMap \quad \pi,\Gamma \vdash isSequential(m) \quad \pi,\Gamma \vdash isGet_m(l_g) \ \wedge \ retv(l_g) \neq \perp}{\pi,\Gamma \vdash \exists \ell_p : isPutter_m(\ell_p, l_g) \wedge arg2(\ell_p) = retv(l_g)}$$

BasicSetAdd

$$\frac{\mathcal{T}_{base}(s) = BasicSet \quad \pi,\Gamma \vdash isSequential(s) \quad \pi,\Gamma \vdash isAdd_s(l_a) \quad \pi,\Gamma \vdash isContains_s(l_c) \quad \pi,\Gamma \vdash l_a \prec l_c \ \wedge \ arg1(l_a) = arg1(l_c)}{\pi,\Gamma \vdash retv(l_c) = true}$$

BasicMapPut

$$\frac{\mathcal{T}_{base}(m) = BasicMap \quad \pi,\Gamma \vdash isSequential(m) \quad \pi,\Gamma \vdash isGet_m(l_g) \quad \pi,\Gamma \vdash isPutter_m(l_p, l_g)}{\pi,\Gamma \vdash arg2(l_p) = retv(l_g)}$$

$$\begin{aligned}
&isContains_o(\ell) \Leftrightarrow \\
&\quad exec(\ell) \ \wedge \ obj(\ell) = o \ \wedge \ name(\ell) = contains \\
&isAdd_o(\ell) \Leftrightarrow \\
&\quad exec(\ell) \ \wedge \ obj(\ell) = o \ \wedge \ name(\ell) = add \\
&isPut_o(\ell) \Leftrightarrow \\
&\quad exec(\ell) \ \wedge \ obj(\ell) = o \ \wedge \ name(\ell) = put \\
&isGet_o(\ell) \Leftrightarrow \\
&\quad exec(\ell) \ \wedge \ obj(\ell) = o \ \wedge \ name(\ell) = get \\
&isPutter_m(\ell_p, \ell_g) \Leftrightarrow \\
&\quad isPut_m(\ell_p) \ \wedge \ arg1(\ell_p) = arg1(\ell_g) \ \wedge \ \ell_p \prec \ell_g \ \wedge \\
&\quad \forall \ell'_p : isPut_m(\ell'_p) \ \wedge \ arg1(\ell'_p) = arg1(\ell_g) \Rightarrow (\ell'_p \preceq \ell_p \ \vee \ \ell_g \prec \ell'_p) \\
&isSequential(o) \Leftrightarrow \\
&\quad \forall \ell, \ell' : (exec(\ell) \ \wedge \ exec(\ell') \ \wedge \ obj(\ell) = o \ \wedge \ obj(\ell') = o) \Rightarrow \\
&\qquad (\ell \preceq \ell' \ \vee \ \ell' \prec \ell)
\end{aligned}$$

Figure 32. Set and Map Inference Rules

BasicMapGet'

$$\frac{\mathcal{T}_{base}(m) = BasicMap \quad \pi,\Gamma \vdash isSequential(m) \quad \pi,\Gamma \vdash isGet_m(l_g) \quad \pi,\Gamma \vdash \neg\exists \ell_p : isPut_m(\ell_p) \wedge arg1(\ell_p) = arg1(l_g) \ \wedge \ \ell_p \prec l_g}{\pi,\Gamma \vdash retv(l_g) = \perp}$$

BasicMapPut'

$$\frac{\mathcal{T}_{base}(m) = BasicMap \quad \pi,\Gamma \vdash isSequential(m) \quad \pi,\Gamma \vdash isGet_m(l_g) \quad \pi,\Gamma \vdash isPut_m(l_p) \quad \pi,\Gamma \vdash arg1(l_p) = arg1(l_g) \ \wedge \ l_p \prec l_g \quad \pi,\Gamma \vdash \forall \ell_p : isPut_m(\ell_p) \Rightarrow arg2(\ell_p) \neq \perp}{\pi,\Gamma \vdash retv(l_g) \neq \perp}$$

Figure 33. Derived Set and Map Inference Rules

The rule BasicMapPut states that if a basic map m is accessed sequentially, for every *get* method call l_g on m, if l_p is the latest preceding *put* method call on m with the same key argument then the value argument of l_p is equal to the return value of l_g.

The derived Set and Map inference rules are presented in Figure 33.

The rule BasicMapGet' states that if a basic map m is accessed sequentially, for every *get* method call l_g on m, if no *put* method call with the same key argument as l_g precedes l_g, then l_g returns $\perp$.

The rule BasicMapPut' states that if a basic map m is accessed sequentially and no *put* method call puts $\perp$ in m, every *get* method call that succeeds a *put* method call with the same key argument does not return $\perp$.

12 Dekker Algorithm

In this section, we prove the mutual exclusion guarantee of the Dekker algorithm using the logic. We presented the Dekker algorithm, π_{Dekker}, in Figure 1.

Theorem 12.1 (Mutual Exclusion)**.**
In every execution of the Dekker specification, at most one thread acquires the lock.
$\forall X \in \mathbb{H}(\pi_{Dekker})\colon (retv_X(L_2) = true) \Rightarrow (retv_X(L_1) = false)$.

Proof.

We show that
(1) $X' \in \mathbb{H}(\pi_{Dekker})$
We show that
(2) $(retv_{X'}(L_2) = true) \Rightarrow (retv_{X'}(L_1) = false)$

By Definition 17 on [1], we have that there exists $\mathcal{X}, X, \sigma, \mathcal{L}$ such that
(3) $\mathcal{X} = (X, \sigma, \mathcal{L}) \in [\![\pi]\!]$
(4) $X' = \sigma(X)$
By Lemma 12.2, we have
(5) $\pi_{Dekker}, \cdot \vdash (retv(L_2) = true) \Rightarrow (retv(L_1) = false)$.
By the soundness theorem, Theorem 13.4, and Definition [13.3] on [5] and [3], we have
(6) $\mathcal{X} \models (retv(L_2) = true) \Rightarrow (retv(L_1) = false)$
By the definition of $\models$ (Figure 8) on [6], [3] and [4], we have
(7) $(retv_{X'}(L_2) = true) \Rightarrow (retv_{X'}(L_1) = false)$. □

Lemma 12.2.
$\pi_{Dekker}, \cdot \vdash (retv(L_2) = true) \Rightarrow (retv(L_1) = false)$.

Proof.

Let
$\pi = \pi_{Dekker}$
We show that
$\pi, \cdot \vdash (retv(L_2) = true) \Rightarrow (retv(L_1) = false)$

Let
(8) $\Gamma = (retv(L_2) = true)$
By rule CondIntro, we have to show that
$\pi, \Gamma \vdash retv(L_1) = false$
By rule Premise on [8], we have
(9) $\pi, \Gamma \vdash retv(L_2) = true$
From π, we have
$cond_\pi(L_2) = true$
Thus,
(10) $\pi, \Gamma \vdash cond_\pi(L_2) = true$
By rule OControl on [10], we have
(11) $\pi, \Gamma \vdash exec(L_2)$
From π, we have
(12) $name_\pi(L_2) = tryLock2$
(13) $R_1 \in Labels(tryLock2)$
From π, we have
$cond_\pi(R_1) = true$
Thus,
(14) $\pi, \Gamma \vdash L_2\text{'}cond_\pi(R_1) = true$
From π, we have
(15) $PreReturns_\pi(R_1) = \emptyset$
By rule IControl on [11]-[15], we have
(16) $\pi, \Gamma \vdash exec(L_2\text{'}R_1)$
By rule Id on [16], we have
(17) $\pi, \Gamma \vdash obj(L_2\text{'}R_1) = f_1$
(18) $\pi, \Gamma \vdash name(L_2\text{'}R_1) = read$
(19) $\pi, \Gamma \vdash retv(L_2\text{'}R_1) = L_2\text{'}x_1$
Similarly, we have
(20) $\pi, \Gamma \vdash exec(L_2\text{'}W_2)$
(21) $\pi, \Gamma \vdash obj(L_2\text{'}W_2) = f_2$
(22) $\pi, \Gamma \vdash name(L_2\text{'}W_2) = write$
(23) $\pi, \Gamma \vdash arg1(L_2\text{'}W_2) = 1$
From the definition of *isRead* on [16], [17] and [18] and rule ConjIntro, we have
(24) $\pi, \Gamma \vdash isRead_{f_1}(L_2\text{'}R_1)$
From rule AReg on [24], we have
(25) $\pi, \Gamma \vdash \exists \ell_W :$
$isWriter_{f_1}(\ell_W, L_2\text{'}R_1) \wedge$
$retv(L_2\text{'}R_1) = arg1(\ell_W)$
Let
(26) $\Gamma' = \Gamma;$
$isWriter_{f_1}(l_W, L_2\text{'}R_1) \wedge$
$arg1(l_W) = retv(L_2\text{'}R_1)$
where l_W is fresh.
By rule Premise on [26], we have
(27) $\pi, \Gamma' \vdash isWriter_{f_1}(l_W, L_2\text{'}R_1)$
(28) $\pi, \Gamma' \vdash arg1(l_W) = retv(L_2\text{'}R_1)$
By rule Id on [11], we have
(29) $\pi, \Gamma \vdash obj(L_2) = this$
(30) $\pi, \Gamma \vdash name(L_2) = tryLock2$
From π, we have
(31) $Returns_\pi(tryLock2) = \{C_{2t}, C_{2f}\}$
By rule Caller on [31], [11], [30], [31], we have
(32) $\pi, \Gamma \vdash$
$(exec(L_2\text{'}C_{2t}) \wedge arg1(L_2\text{'}C_{2t}) = retv(L_2)) \vee$
$(exec(L_2\text{'}C_{2f}) \wedge arg1(L_2\text{'}C_{2f}) = retv(L_2))$
We apply rule DisjElim to [32]:

Right:
Let
(33) $\Gamma' = \Gamma;$
$(exec(L_2\text{'}C_{2f}) \wedge arg1(L_2\text{'}C_{2f}) = retv(L_2))$
By rule Premise on [33], we have
(34) $\pi, \Gamma' \vdash exec(L_2\text{'}C_{2f})$

(35) $\pi, \Gamma' \vdash arg1(L_2\text{'}C_{2f}) = retv(L_2)$

From π, we have

(36) $arg1(C_{2f}) = false$

By rule Id on [34], [36], we have

(37) $\pi, \Gamma' \vdash arg1(L_2\text{'}C_{2f}) = false$

From rule ETrans and rule ESym on [35], and [37], we have

(38) $\pi, \Gamma' \vdash retv(L_2) = false$

By weakening (Lemma 13.2) on [33] [9], we have

(39) $\pi, \Gamma' \vdash retv(L_2) = true$

By rule NegElim on [38] and [39], we have

(40) $\pi, \Gamma' \vdash retv(L_1) = false$

Left:

Let

(41) $\Gamma' = \Gamma;$
$(exec(L_2\text{'}C_{2t}) \ \wedge \ arg1(L_2\text{'}C_{2t}) = retv(L_2))$

By rule Premise on [41], we have

(42) $\pi, \Gamma' \vdash exec(L_2\text{'}C_{2t})$

(43) $\pi, \Gamma' \vdash arg1(L_2\text{'}C_{2t}) = retv(L_2)$

From π, we have

(44) $cond_\pi(C_{2t}) = (x_1 = 0)$

By rule IControl on [42] and [44] we have

(45) $\pi, \Gamma' \vdash (L_2\text{'}x_1 = 0)$

From [28], [19], [45], weakening (Lemma 13.2) and rule ETrans, we have

(46) $\pi, \Gamma' \vdash arg1(l_W) = 0$

From the definition of $isWriter$ on [27] and rule ConjElimL and rule ConjElimR, we have

(47) $\pi, \Gamma' \vdash obj(l_W) = f_1$

(48) $\pi, \Gamma' \vdash name(l_W) = write$

(49) $\pi, \Gamma' \vdash exec(l_W)$

(50) $\pi, \Gamma' \vdash l_W \prec_{f_1} L_2\text{'}R_1$

(51) $\pi, \Gamma' \vdash \forall \ell_{W'} : isWrite_{f_1}(\ell_{W'}) \Rightarrow$
$\ell_{W'} \preceq_{f_1} l_W \ \vee \ L_2\text{'}R_1 \prec_{f_1} \ell_{W'}$

From the definition of π, we have

(52) $calls_\pi(f_1, write) = \{W_1, W_{01}\}$

From rule Src on [47], [48], [49] and [52], we have that for some fresh ς

(53) $\pi, \Gamma' \vdash l_W = \varsigma\text{'}W_1 \ \vee \ l_W = \varsigma\text{'}W_{01}$

We apply rule DisjElim to [53]:

Left:

(54) $\Gamma'' = \Gamma';$
$l_W = \varsigma\text{'}W_1$

From [49], [54], weakening (Lemma 13.2), we have

(55) $\pi, \Gamma'' \vdash exec(\varsigma\text{'}W_1)$

From π, we have

(56) $arg1_\pi(W_1) = 1$

By rule Id on [54], [56], we have

(57) $\pi, \Gamma'' \vdash arg1(\varsigma\text{'}W_1) = 1$

From [54], [57], we have

(58) $\pi, \Gamma'' \vdash arg1(l_W) = 1$

By weakening (Lemma 13.2) on [46], we have

(59) $\pi, \Gamma'' \vdash arg1(l_W) = 0$

By rule ETrans and rule ESym on [58], [59], we have

(60) $\pi, \Gamma'' \vdash 0 = 1$

By rule NegElim on rule Zero and [60], we have

(61) $\pi, \Gamma'' \vdash retv(L_1) = false$

Right:

(62) $\Gamma'' = \Gamma';$
$l_W = \varsigma\text{'}W_{01}$

By rule Premise on [62], we have

(63) $\pi, \Gamma'' \vdash l_W = \varsigma\text{'}W_{01}$

From π, we have

(64) $W_{01} \in Labels_\pi(init)$

By rule Callee on [63] and [64] we have

(65) $\pi, \Gamma'' \vdash \neg(\varsigma = \epsilon)$

(66) $\pi, \Gamma'' \vdash exec(\varsigma)$

(67) $\pi, \Gamma'' \vdash obj(\varsigma) = this$

(68) $\pi, \Gamma'' \vdash name(\varsigma) = init$

From π, we have

(69) $Calls_\pi(this, init) = \{L_0\}$

By rule Src on [65]-[69] we have

(70) $\pi, \Gamma'' \vdash \varsigma = L_0$

From [63] and [70], we have

(71) $\pi, \Gamma'' \vdash l_W = L_0\text{'}W_{01}$

From π, we have

$cond_\pi(L_1) = true$

Thus,

(72) $\pi, \Gamma'' \vdash cond_\pi(L_1) = true$

By rule OControl on [72], we have

(73) $\pi, \Gamma'' \vdash exec(L_1)$

From π, we have

(74) $name_\pi(L_1) = tryLock1$

(75) $R_2 \in Labels(tryLock1)$

From π, we have

$cond_\pi(R_2) = true$

Thus,

(76) $\pi, \Gamma'' \vdash L_1\text{'}cond_\pi(R_2) = true$

From π, we have

(77) $PreReturns_\pi(R_2) = \emptyset$

By rule IControl on [73]-[77], we have

(78) $\pi, \Gamma'' \vdash exec(L_1\text{'}R_2)$

From π we have

(79) $obj_\pi(R_2) = f_2$

(80) $name_\pi(R_2) = read$

(81) $retv_\pi(R_2) = x_2$

By rule Id on [78] and [79]-[81], and then rule ConjElimL and rule ConjElimR, we have

(82) $\pi, \Gamma'' \vdash obj(L_1\text{'}R_2) = f_2$

(83) $\pi, \Gamma'' \vdash name(L_1\text{'}R_2) = read$

(84) $\pi, \Gamma'' \vdash retv(L_1\text{'}R_2) = L_1\text{'}x_2$

From the definition of $isRead$ on [78], [82], [83] and rule ConjIntro, we have

(85) $\pi, \Gamma'' \vdash isRead_{f_2}(L_1\text{'}R_2)$

Similarly, we have that
(86) $\pi, \Gamma'' \vdash exec(L_1\text{'}W_1)$
(87) $\pi, \Gamma'' \vdash obj(L_1\text{'}W_1) = f_1$
(88) $\pi, \Gamma'' \vdash name(L_1\text{'}W_1) = write$
(89) $\pi, \Gamma'' \vdash arg1(L_1\text{'}W_1) = 1$
(90) $\pi, \Gamma'' \vdash isWrite_{f_1}(L_1\text{'}W_1)$
By rule UnivElim on [51], and [90], we have
(91) $\pi, \Gamma'' \vdash L_1\text{'}W_1 \preceq_{f_1} l_W \vee L_2\text{'}R_1 \prec_{f_1} L_1\text{'}W_1$
By rule LSubs on [91] and [71], we have
(92) $\pi, \Gamma'' \vdash$
$L_1\text{'}W_1 \preceq_{f_1} L_0\text{'}W_{01} \vee$
$L_2\text{'}R_1 \prec_{f_1} L_1\text{'}W_1$
From π, we have
(93) $L_0 \rightarrow_\pi L_1$
By rule LSubs on [66] and [70], we have
(94) $\pi, \Gamma'' \vdash exec(L_0)$
By rule P2X on [93], [94] and [73], we have
(95) $\pi, \Gamma'' \vdash L_0 \prec L_1$
By rule LSubs on [49] and [71], we have
(96) $\pi, \Gamma'' exec(L_0\text{'}W_{01})$
By rule OX2IX on [95], and [96], and [86], we have
(97) $\pi, \Gamma'' \vdash L_0\text{'}W_{01} \prec L_1\text{'}W_1$
By rule Id on [96], we have
(98) $\pi, \Gamma'' \vdash obj(L_0\text{'}W_{01}) = f_1$
By rule X2L on [97], [98] and [87], we have
(99) $\pi, \Gamma'' \vdash L_0\text{'}W_{01} \prec_{f_1} L_1\text{'}W_1$
By rule LASym on [99], and rule ConjElimL, we have
(100) $\pi, \Gamma'' \vdash \neg(L_1\text{'}W_1 \prec_{f_1} L_0\text{'}W_{01})$
By rule DisjSyllL on [92], [100], we have
(101) $\pi, \Gamma'' \vdash L_2\text{'}R_1 \prec_{f_1} L_1\text{'}W_1$
From π, we have
(102) $W_2 \rightarrow_\pi R_1$
From rule P2X on [102], [20], [16], and weakening (Lemma 13.2), we have
(103) $\pi, \Gamma'' \vdash L_2\text{'}W_2 \prec L_2\text{'}R_1$
From π, we have
(104) $W_1 \rightarrow_\pi R_2$
From rule P2X on [104], [86] and [78], we have
(105) $\pi, \Gamma'' \vdash L_1\text{'}W_1 \prec L_1\text{'}R_2$
From rule XLTrans on [103], [101] and [105], we have
(106) $\pi, \Gamma'' \vdash L_2\text{'}W_2 \prec L_1\text{'}R_2$
From rule X2L on [106], [21] and [82], we have
(107) $\pi, \Gamma'' \vdash L_2\text{'}W_2 \prec_{f_2} L_1\text{'}R_2$
We show that
(108) $\pi, \Gamma'' \vdash \forall \ell_W :$
$isWrite_{f_2}(\ell_W) \Rightarrow$
$\ell_W \preceq_{f_2} L_2\text{'}W_2 \vee L_1\text{'}R_2 \prec_{f_2} \ell_W$
Let
(109) $\Gamma''' = \Gamma''; isWrite_{f_2}(l'_W)$
By rule UnivIntro and rule CondIntro, we have to show that
$\pi, \Gamma''' \vdash l'_W \preceq_{f_2} L_2\text{'}W_2 \vee L_1\text{'}R_2 \prec_{f_2} l'_W$
By rule Premise on [109], we have
(110) $\pi, \Gamma''' \vdash isWrite_{f_2}(l'_W)$
From definition of $isWrite$ on [110], we have
(111) $\pi, \Gamma''' \vdash$
$obj(l'_W) = f_2 \wedge$
$name(l'_W) = write \wedge$
$exec(l'_W)$
From the definition of π, we have
(112) $calls_\pi(f_2, write) = \{W_{02}, W_2\}$
By rule Src on [111] and [112], we have that for some fresh ς,
(113) $\pi, \Gamma''' \vdash l'_W = \varsigma\text{'}W_{02} \vee l'_W = \varsigma\text{'}W_2$
We apply rule DisjElim on [113]:

Left:
(114) $\Gamma'''' = \Gamma'''; (l'_W = \varsigma\text{'}W_{02})$
By rule Premise on [114], we have
(115) $\pi, \Gamma'''' \vdash l'_W = \varsigma\text{'}W_{02}$
By rule LSubs on [111], [115] and weakening (Lemma 13.2), we have
(116) $\pi, \Gamma'''' \vdash exec(\varsigma\text{'}W_{02})$
From π, we have
(117) $W_{02} \in Labels_\pi(init)$
By rule Callee on [116] and [117], we have
(118) $\pi, \Gamma'''' \vdash \neg(\varsigma = \epsilon)$
(119) $\pi, \Gamma'''' \vdash exec(\varsigma)$
(120) $\pi, \Gamma'''' \vdash obj(\varsigma) = this$
(121) $\pi, \Gamma'''' \vdash name(\varsigma) = init$
From π, we have
(122) $calls_\pi(this, init) = \{L_0\}$
By rule Src on [118]-[122], we have
(123) $\pi, \Gamma'''' \vdash \varsigma = L_0$
By rule LSubs on [115], [123], we have
(124) $\pi, \Gamma'''' \vdash l'_W = L_0\text{'}W_{02}$
By rule LSubs on [111], [124], we have
(125) $\pi, \Gamma''' \vdash obj(L_0\text{'}W_{02}) = f_2$
(126) $\pi, \Gamma''' \vdash exec(L_0\text{'}W_{02})$
From π, we have
(127) $L_0 \rightarrow_\pi L_2$
By rule P2X on [127], [94] and [11], weakening (Lemma 13.2), we have
(128) $\pi, \Gamma'''' \vdash L_0 \prec L_2$
By rule OX2IX on [128], and [126], and [20], we have
(129) $\pi, \Gamma'''' \vdash L_0\text{'}W_{02} \prec L_2\text{'}W_2$
By rule X2L on [129], and [125], and [21], we have
(130) $\pi, \Gamma'''' \vdash L_0\text{'}W_{02} \prec_{f_2} L_2\text{'}W_2$
By rule DisjIntroL on [130], we have
(131) $\pi, \Gamma''' \vdash$
$L_0\text{'}W_{02} \preceq_{f_2} L_2\text{'}W_2 \vee$
$L_1\text{'}R_2 \prec_{f_2} L_0\text{'}W_{02}$
By rule LSubs on [131] and [124], we have
(132) $\pi, \Gamma''' \vdash$

$$l'_W \preceq_{f_2} L_2\text{'}W_2 \vee \\ L_1\text{'}R_2 \prec_{f_2} l'_W$$

Right:

(133) $\Gamma'''' = \Gamma'''; (l'_W = \varsigma\text{'}W_2)$

By rule Premise on [133], we have

(134) $\pi, \Gamma'''' \vdash l'_W = \varsigma\text{'}W_2$

Similar to the previous part, we can show that

(135) $\pi, \Gamma'''' \vdash \varsigma = L_2$

By rule LSubs on [134] and [135], we have

(136) $\pi, \Gamma'''' \vdash l'_W = L_2\text{'}W_2$

By rule DisjIntroR on [136], we have

(137) $\pi, \Gamma''' \vdash l'_W \preceq_{f_2} L_2\text{'}W_2$

Thus, by rule DisjIntroL on [137], we have

$\pi, \Gamma''' \vdash$
$l'_W \preceq_{f_2} L_2\text{'}W_2 \vee$
$L_1\text{'}R_2 \prec_{f_2} l'_W$

By rule ConjIntro and the definition of $isWrite$ on [20]-[22] and weakening (Lemma 13.2), we have

(138) $\pi, \Gamma'' \vdash isWrite_{f_2}(L_2\text{'}W_2)$

By rule ConjIntro and the definition of $isWriter$ on [138], [107], and [108], we have

(139) $\pi, \Gamma'' \vdash isWriter_{f_2}(L_2\text{'}W_2, L_1\text{'}R_2)$

By rule ConjIntro and the definition of $isRead$ on [78], [82] and [83], we have

(140) $\pi, \Gamma'' \vdash isRead_{f_2}(L_1\text{'}R_2)$

From rule AReg' on [140] and [139], we have

(141) $\pi, \Gamma'' \vdash retv(L_1\text{'}R_2) = arg1(L_2\text{'}W_2)$

By rule ETrans and rule ESym on [141], [84] and [23], we have

(142) $\pi, \Gamma'' \vdash L_1\text{'}x_2 = 1$

By rule Zero and rule ESubs on [142], we have

(143) $\pi, \Gamma'' \vdash \neg(L_1\text{'}x_2 = 0)$

From π, we have that

(144) $cond_\pi(C_{1f}) = \neg(x_2 = 0)$

(145) $name_\pi(L_1) = tryLock1$

(146) $C_{1f} \in Labels_\pi(tryLock1)$

(147) $PreReturns_\pi(C_{1f}) = \emptyset$

From [144], we have

(148) $L_1\text{'}cond_\pi(C_{1f}) = \neg(L_1\text{'}x_2 = 0)$

From [143] and [148], we have

(149) $\pi, \Gamma'' \vdash L_1\text{'}cond_\pi(C_{1f})$

By rule IControl on [73], [146], [145], [149], [147] we have

(150) $\pi, \Gamma'' \vdash exec(L_1\text{'}C_{1f})$

From π, we have that

(151) $C_{1f} \in Returns_\pi(tryLock1)$

(152) $arg1_\pi(C_{1f}) = false$

By rule Id on [150] and [152], we have

(153) $\pi, \Gamma'' \vdash arg1(L_1\text{'}C_{1f}) = false$

By rule Ret on [150], [151], we have

(154) $\pi, \Gamma'' \vdash retv(L_1) = arg1(L_1\text{'}C_{1f})$

By rule ETrans and rule ESym on [153], [154], we have

$\pi, \Gamma'' \vdash retv(L_1) = false$

□

13 Soundness

In this section, we present the soundness, exchange, and weakening lemmas for the logic.

The semantics satisfies the classical exchange and weakening lemmas.

Lemma 13.1 (Exchange)**.**
$\forall \pi, \Gamma, \Gamma', \mathcal{A}, \mathcal{A}', \mathcal{A}'':$
$(\pi, \Gamma; \mathcal{A}; \mathcal{A}'; \Gamma' \vdash \mathcal{A}'') \Rightarrow (\pi, \Gamma; \mathcal{A}'; \mathcal{A}; \Gamma' \vdash \mathcal{A}'')$

Lemma 13.2 (Weakening)**.**
$\forall \pi, \Gamma, \mathcal{A}, \mathcal{A}':$
$(\pi, \Gamma \vdash \mathcal{A}) \Rightarrow (\pi, \Gamma; \mathcal{A}' \vdash \mathcal{A})$

To define the soundness, we first define the models relation between specifications and assertions.

Definition 13.3. A specification π models an assertion $\mathcal{A}$ if and only if every execution of π models $\mathcal{A}$.
$\pi \models \mathcal{A}$ iff $\forall \mathcal{X} \in [\![\pi]\!] : \mathcal{X} \models \mathcal{A}$

The logic is sound. The following theorem states that the logic derives valid conclusions from valid premises.

Theorem 13.4 (Soundness)**.**
$\forall \pi, \mathcal{A} : ((\pi, \Gamma \vdash \mathcal{A}) \ \wedge \ (\pi \models \Gamma)) \Rightarrow (\pi \models \mathcal{A})$.

See Section 15.2 for the proof.

14 Client Assertions

$$\Gamma_0 = \Gamma_1 \ \wedge \ \Gamma_2 \ \wedge \ \Gamma_3 \ \wedge \ \Gamma_4 \ \wedge \ \Gamma_5 \ \wedge \ \Gamma_6 \ \wedge \ \Gamma_7 \tag{136}$$

$$\Gamma_1 = \forall t : \mathit{Let}\ l = \mathit{initOf}(t) : \mathit{isInit}(l) \ \wedge \ \mathit{thread}(l) = t \tag{137}$$

$$\Gamma_2 = \forall \ell, \ell' : (\mathit{isInit}(\ell) \ \wedge \ \mathit{isInit}(\ell') \ \wedge \ \mathit{thread}(\ell) = \mathit{thread}(\ell')) \Rightarrow \ell = \ell' \tag{138}$$

$$\Gamma_3 = \forall \ell, \ell' : (\mathit{isInit}(\ell) \ \wedge \ \mathit{exec}(\ell') \ \wedge \ \mathit{obj}(\ell') = \mathit{this} \ \wedge \ \mathit{thread}(\ell) = \mathit{thread}(\ell')) \Rightarrow \ell \preceq \ell' \tag{139}$$

$$\Gamma_4 = \forall t : \mathit{Let}\ l = \mathit{commitOf}(t) : \mathit{isCommitted}(t) \Rightarrow (\mathit{isCommit}(l) \ \wedge \ \mathit{thread}(l) = t) \tag{140}$$

$$\Gamma_5 = \forall \ell, \ell' : (\mathit{isCommit}(\ell) \ \wedge \ \mathit{isCommit}(\ell') \ \wedge \ \mathit{thread}(\ell) = \mathit{thread}(\ell')) \Rightarrow \ell = \ell' \tag{141}$$

$$\Gamma_6 = \forall \ell, \ell' : (\mathit{exec}(\ell) \ \wedge \ \mathit{obj}(\ell) = \mathit{this} \ \wedge \ \mathit{isCommit}(\ell') \ \wedge \ \mathit{thread}(\ell) = \mathit{thread}(\ell')) \Rightarrow \ell \preceq \ell' \tag{142}$$

$$\Gamma_7 = \forall t : \mathit{isCommitted}(t) \ \vee \ \mathit{isAborted}(t) \tag{143}$$

Figure 34. Properties of Well-formed Client Transactions

Any client program must satisfy seven conditions that we will specify with the help of the definitions in Figure 9.(a). Figure 34 shows the seven conditions. Γ_1: Every transaction is initialized. Γ_2: Every transaction is initialized only once. Γ_3: The initialization operation of each transaction is executed before its other operations. Γ_4: If a transaction is committed, it executed the commit operation. Γ_5: Every transaction executes the commit operation at most once. Γ_6: The commit operation of each transaction is executed after its other operations. Γ_7: Each transaction is either aborted or committed.

The following lemma states that these properties of client transactions are valid for every TM algorithm specification.

Lemma 14.1. $\forall \pi \in \Pi_{TM} : \pi \models \Gamma_0$.

See section 15.4 for the proof.

15 Proofs

15.1 Semantics

15.1.1 Execution History

Lemma 10.1:

We Assume

(1) $l \prec_X l'$

From [1] and definition of $\sim_X$, we have

(2) $\neg(l' \sim_X l)$

From [1], we have

(3) $rEv(l) \lhd_X iEv(l')$

As X is a valid history, we have

(4) $iEv(l) \lhd_X rEv(l)$

(5) $iEv(l') \lhd_X rEv(l')$

From [4], [3], and [5], we have

(6) $iEv(l) \lhd_X rEv(l')$

From [6], we have

(7) $\neg(rEv(l') \lhd_X iEv(l))$

From [7], and definition of $\prec_X$, we have

(8) $\neg(l' \prec_X l)$

From [3] and [7], we have

(9) $\neg(l' = l)$

Lemma 10.2:

Straightforward from the definition of $\prec_X$.

Lemma 10.3:

We have

(1) $l_1 \prec_X l_2$

(2) $l_3 \prec_X l_4$

(3) $l_2 \sim_X l_3$

From [1], we have

(4) $rEv(l_1) \lhd_X iEv(l_2)$

From [2], we have

(5) $rEv(l_3) \lhd_X iEv(l_4)$

From [3], we have

(6) $\neg(l_3 \prec_X l_2)$

From [6], we have

(7) $\neg(rEv(l_3) \lhd_X iEv(l_2))$

From [7], we have

(8) $iEv(l_2) \lhd_X rEv(l_3)$

From [4], [8], and [5], we have

(9) $rEv(l_1) \lhd_X iEv(l_4)$

From [9], we have

$l_1 \prec_X l_4$

Lemma 10.4:

Straightforward from the definition of $\prec_X$ and $\sim_X$.

Lemma 10.5:

Straightforward from the definition of $\prec_X$.

Lemma 10.6:

Straightforward from the definition of $\prec_X$ and $\lhd_X$.

Lemma 10.7:
Straightforward from the definition of $\prec_X$ and $\lhd_X$.

15.1.2 Synchronization Object Types

Lemma 10.11:
Straightforward from $\prec_X \subseteq \prec_L$.

Lemma 10.12:
Straightforward from Lemmas 10.16, [10.4], 10.11, and 10.13.

Lemma 10.13:
We have
(1) $l \prec_L l'$
From [1], we have
(2) $rEv(l) \lhd_L iEv(l')$
From the well-formedness of the history O,
we have
(3) $iEv(l) \lhd_L rEv(l)$
(4) $iEv(l') \lhd_L rEv(l')$
From [3], [2] and [4], we have
(5) $iEv(l) \lhd_L rEv(l')$
From [5], we have
(6) $\neg(rEv(l') \lhd_L iEv(l))$
From [2] and [6], we have
(7) $\neg(l' = l)$
From the definition of $\prec_X$ on [6], we have
(8) $\neg(l' \prec_L l)$
The conclusion is
[8] and [7]

Lemma 10.14:
Straightforward from the fact that L is a member of sequential specification and
a sequential specification is a set of sequential histories and
the execution order is total in sequential histories.

Lemma 10.15:
Straightforward from the fact that L is a member of sequential specification and
a sequential specification is a set of sequential histories and
the execution order is total in sequential histories.
We have
(1) $l \in X$
(2) $l' \in X$
(3) $X \equiv L$
(4) $L \in SeqSpec(o)$
From [4], we have
(5) $L \in Sequential$
From [3], [1] and [2], we have
(6) $l \in L$
(7) $l' \in L$
From [4], [6] and [7], we have
$l \prec_L l' \ \vee \ l' \prec_L l \ \vee \ l = l'$

Lemma 10.16:
Straightforward from the fact that L is equivalent to X.
We have
(1) $X \equiv L$
(2) $L \in SeqSpec(o)$
(3) $l \prec_L l'$
From [3], we have
(4) $l \in L$
(5) $l' \in L$
From [2] on [4] and [5], we have
(6) $obj_L(l) = o$
(7) $obj_L(l') = o$
From [1] on [4] and [5], we have
$l \in X$
$l' \in X$
From [1] on [6] and [7], we have
$obj_X(l) = o$
$obj_X(l') = o$

Lemma 10.17:
Using L2X and XTotal, we have four cases:
Case: $l \prec l'$
Straightforward from XTrans.
Case: $l \sim l'$
Straightforward from XXTrans.
Case: $l' \prec l$
Straightforward from X2L and LASym.
Case: $l' = l$
Straightforward from LASym.

Lemma 10.19:
Derived from the semantics of basic objects (Definition 10.8) and the sequential specification of register (Definition 10.18).

Lemma 10.21:
Derived from the semantics of basic register (Definition 10.20).

Lemma 10.22:
This is a restatement of Theorem 3 from the original definition of linearizability [19]. Derivable from the semantics of linearizable objects (Definition 10.10) and the sequential specification of register (Definition 10.18).

Lemma 10.24:
Derivable from the semantics of linearizable objects (Definition 10.10) and the sequential specification of cas register (Definition 10.23).

Lemma 10.25:
Derivable from the semantics of linearizable objects (Definition 10.10) and the sequential specification of cas register (Definition 10.23).

Lemma 10.28:
Derivable from the semantics of linearizable objects (Definition 10.10), the sequential specification of the lock (Definition 10.26), the owner-respecting property (Definition 10.27), and that the sub-history for each thread is sequential (from the definition of execution histories).

Lemma 10.29:
Derived from Lemma 10.28.

Lemma 10.30:
Derived from Lemma 10.28 and the sequential specification of lock (Definition 10.26).

Lemma 10.31:
Derived from Lemma 10.28 and the sequential specification of lock (Definition 10.26).

Lemma 10.32:
Derived from Lemma 10.28 and the sequential specification of lock (Definition 10.26).

Lemma 10.34:
Derivable from the semantics of linearizable objects (Definition 10.10), the sequential specification of the lock (Definition 10.33), the owner-respecting property (Definition 10.34), and that the sub-history for each thread is sequential (from the definition of execution histories).

Lemma 10.35:
Derived from Lemma 10.34.

Lemma 10.36:
Derived from Lemma 10.34 and the sequential specification of try-lock (Definition 10.33).

Lemma 10.37:
Derived from Lemma 10.34 and the sequential specification of try-lock (Definition 10.33).

Lemma 10.38:
Derived from Lemma 10.34 and the sequential specification of try-lock (Definition 10.33).

Lemma 10.41:
Derivable from the semantics of linearizable objects (Definition 10.10), the sequential specification of counter (Definition 10.40).

Lemma 10.43:
Derivable from the semantics of basic objects (Definition 10.8), the sequential specification of set (Definition 10.42).

Lemma 10.44:
Derivable from the semantics of basic objects (Definition 10.8), the sequential specification of set (Definition 10.42).

Lemma 10.46:
Derivable from the semantics of basic objects (Definition 10.8), the sequential specification of set (Definition 10.45).

Lemma 10.47:
Derivable from the semantics of basic objects (Definition 10.8), the sequential specification of set (Definition 10.45).

15.2 Soundness

Theorem 13.4 (Soundness).
$\forall \pi, \mathcal{A}\colon ((\pi, \Gamma \vdash \mathcal{A}) \ \wedge \ (\pi \models \Gamma)) \Rightarrow (\pi \models \mathcal{A})$.

Proof.

Hypothesis
(1) $\pi, \Gamma \vdash \mathcal{A}$
(2) $\mathcal{X} \models \Gamma$
Desired Conclusion
$\pi \models \mathcal{A}$

Let
(3) $\pi = (\mathcal{T}, \mathcal{D}, \mathcal{P})$
(4) $\mathcal{D} = d^*$
(5) $\mathcal{P} = p_0, (p_1 || p_2 || ... || p_n)$
(6) $\mathcal{X} = (X, \sigma, \mathcal{L}) \in [\![\pi]\!]$
By Definitions 13.3, we need to show that
$\mathcal{X} \models \mathcal{A}$

Let
(7) $X' = \sigma(X)$
By definition [17] on [6] and [7], we have
(8) $X' \in \mathbb{H}(\pi)$
By definition [13] on [6], we have
(9) $\forall o\colon \mathcal{T}_{base}(o) \in BT \Rightarrow X'|o \in \mathbb{H}_B(o)$
(10) $\forall o\colon \mathcal{T}_{base}(o) \in LT \Rightarrow (X'|o, \mathcal{L}(o)) \in \mathbb{H}_L(o)$
$\exists X_1, ..., X_n\colon$
(11) $\forall i \in \{0..n\}\colon (X_i, \sigma) \in [\![p_i]\!] \wedge$
(12) $X'' \in Interleave(X_1, \ldots, X_n) \wedge X = X_0 \cdot X''$
By definition [85] on [9], we have
(13) $\forall o\colon o \in \mathcal{T}_{base}(o) \in BT \Rightarrow (X'|o \in Sequential) \Rightarrow (X'|o \in SeqSpec(o))$
By definition [87] on [10], we have
(14) $\forall o\colon o \in \mathcal{T}_{base}(o) \in LT \Rightarrow X'|o \equiv \mathcal{L}(o) \wedge \mathcal{L}(o) \in SeqSpec(o) \wedge \prec_{X'|o} \subseteq \prec_{\mathcal{L}(o)}$

Induction on the derivation of [1]:
Case rule X2L:
By rule X2L on [1], we have that
(15) $\mathcal{T}_{base}(o) \in LT$
(16) $\pi, \Gamma \vdash l \prec l'$
(17) $\pi, \Gamma \vdash obj(l) = obj(l') = o$
(18) $\mathcal{A} = l \prec_o l'$
We show that
$\mathcal{X} \models \mathcal{A}$
That is
$l \prec_{\mathcal{L}(\sigma(o))} l'$

By the induction hypothesis on [16] and [17], and then [2], [6] and [7], we have
(19) $l \prec_{X'} l'$
(20) $obj_{X'}(l) = obj_{X'}(l') = \sigma(o)$
From [19] and [20], we have
(21) $l \prec_{X'|\sigma(o)} l'$
By [15], we have
(22) $\mathcal{T}_{base}(\sigma(o)) \in LT$
By [10] and [22], we have
(23) $(X'|\sigma(o), \mathcal{L}(\sigma(o))) \in \mathbb{H}_L(o)$
By Lemma 10.11 on [23] and [21], we have
$l \prec_{\mathcal{L}(\sigma(o))} l'$

Case rule Src:
We have that
(24) $\mathcal{A} = \bigvee_{i=1..n} c = c_i$
(25) $\pi, \Gamma \vdash exec(\varsigma\text{'}c)$
(26) $\pi, \Gamma \vdash obj(\varsigma\text{'}c) = \theta$
(27) $\pi, \Gamma \vdash name(\varsigma\text{'}c) = n$
(28) $Calls_\pi(basename(\theta), n) = \{\overline{c_i}\}$
We show that
$\mathcal{A} \models \bigvee_{i=1..n} c = c_i$
that is
$\bigvee_{i=1..n} c = c_i$

By the induction hypothesis on [25], [26], [27], and then [2], [6] and [7], we have
(29) $\varsigma\text{'}c \in X'$
(30) $obj_{X'}(\varsigma\text{'}c) = \varsigma\text{'}\theta$
(31) $name_{X'}(\varsigma\text{'}c) = n$
From [7] and [12] on [29], [30], [31], we have
$\exists i \in 0..n\colon$
(32) $\varsigma\text{'}c \in X_i$
(33) $obj_{X_i}(\varsigma\text{'}c) = \varsigma\text{'}\theta$
(34) $name_{X_i}(\varsigma\text{'}c) = n$
By Lemma 15.2 on [11] and [32], we have
(35) $basename(obj_{X_i}(\varsigma\text{'}c)) = obj_\pi(c)$
(36) $name_{X_i}(\varsigma\text{'}c) = name_\pi(c)$
By the definition of *basename* and ', we have
(37) $basename(\varsigma\text{'}\theta) = basename(\theta)$
From [35], [30] and [37], we have
(38) $basename(obj_\pi(c)) = basename(\theta)$
From [36] and [34], we have
(39) $name_\pi(c) = n$
From the definition of $calls_\pi(basename(\theta), n)$

on [38] and [39], we have
(40) $c \in calls_\pi(basename(\theta), n)$
From [28] and [36], we have
$\bigvee_{i=1..n} c = c_i$

Case rule P2X:
We have that
(41) $\mathcal{A} = \varsigma\text{'}c_1 \preceq \varsigma\text{'}c_2$
(42) $c_1 \rightarrow_\pi c_2$
(43) $\pi, \Gamma \vdash exec(\varsigma\text{'}c_1)$
(44) $\pi, \Gamma \vdash exec(\varsigma\text{'}c_2)$
We show that
$\mathcal{X} \models \varsigma\text{'}c_1 \prec \varsigma\text{'}c_2$
that is
$\varsigma\text{'}c_1 \prec_{X'} \varsigma\text{'}c_2$

By the induction hypothesis on [43], [44], and then [2], we have
(45) $\mathcal{X} \models exec(\varsigma\text{'}c_1)$
(46) $\mathcal{X} \models exec(\varsigma\text{'}c_2)$
that is
(47) $\varsigma\text{'}c_1 \in X'$
(48) $\varsigma\text{'}c_2 \in X'$
From Lemma 15.6 on [8], [47] and [48] [42], we have
$\varsigma\text{'}c_1 \prec_{X'} \varsigma\text{'}c_2$

Case rule OX2IX:
We have that
(49) $\mathcal{A} = c_1\text{'}c_3 \prec c_2\text{'}c_4$
(50) $\pi, \Gamma \vdash c_1 \prec c_2$
(51) $\pi, \Gamma \vdash exec(c_1\text{'}c_3)$
(52) $\pi, \Gamma \vdash exec(c_2\text{'}c_4)$
We show that
$\mathcal{X} \models c_1\text{'}c_3 \prec c_2\text{'}c_4$
that is
$c_1\text{'}c_3 \prec_{X'} c_2\text{'}c_4$

By the induction hypothesis on [50], [51], [52], and then [2], we have
(53) $\mathcal{X} \models c_1 \prec c_2$
(54) $\mathcal{X} \models exec(c_1\text{'}c_3)$
(55) $\mathcal{X} \models exec(c_2\text{'}c_4)$
that is
(56) $c_1 \prec_{X'} c_2$
(57) $c_1\text{'}c_3 \in X'$
(58) $c_2\text{'}c_4 \in X'$
From [56], we have
(59) $rEv(c_1) \lhd_{X'} iEv(c_2)$
From Lemma 15.7 on [8] and [57], we have
(60) $rEv(c_1\text{'}c_3) \lhd_{X'} rEv(c_1))$
From Lemma 15.7 on [8], and [58], we have
(61) $(iEv(c_2) \lhd_{X'} iEv(c_2\text{'}c_4)$
From [60], [59] and [61], we have
(62) $rEv(c_1\text{'}c_3) \lhd_{X'} iEv(c_2\text{'}c_4)$
From [62], we have
(63) $c_1\text{'}c_3 \prec_{X'} c_2\text{'}c_4$

Case rule IControl:
We have that
(64) $\mathcal{A} =$
$exec(c\text{'}c') \Leftrightarrow$
$exec(c) \wedge$
$\bigvee_{c_i} c' = c_i \wedge$
$c\text{'}cond_\pi(c') \wedge$
$\bigwedge_{i=1..n} \neg exec(c\text{'}c_i)$
(65) $Labels(name_\pi(c)) = \{\overline{c_i}\}$
(66) $PreReturns_\pi(c') = \{\overline{c_r}\}$
We show that
$\mathcal{X} \models \mathcal{A}$
That is
$c\text{'}c' \in X' \Leftrightarrow$
$c \in X' \wedge$
$\bigvee_{c_i} c' = c_i \wedge$
$\sigma(c\text{'}cond_\pi(c')) \wedge$
$\bigwedge_{c_r} \neg(c\text{'}c_r \in X')$

We first show that
$c\text{'}c' \in X' \Rightarrow$
$c \in X' \wedge$
$\bigvee_{c_i} c' = c_i \wedge$
$\sigma(c\text{'}cond_\pi(c')) \wedge$
$\bigwedge_{i=1..n} \neg(c\text{'}c_i \in X')$
We assume that
(67) $c\text{'}c' \in X'$
We show that
$c \in X' \wedge$
$\bigvee_{c_i} c' = c_i \wedge$
$\sigma(c\text{'}cond_\pi(c')) \wedge$
$\bigwedge_{i=1..n} \neg(c\text{'}c_i \in X')$
From [7] and [12] on [67], we have
$\exists i \in \{0..n\}$:
(68) $c\text{'}c' \in X_i$
By Lemma 15.3 on [65], [66], [11] and [68], we have
(69) $c \in X_i \wedge$
$\bigvee_{c_i} c' = c_i \wedge$
$\sigma(c\text{'}cond_\pi(c')) \wedge$
$\bigwedge_{i=1..n} \neg(c\text{'}c_i \in X_i)$
From [7] and [12] and uniqueness of label c on [69], we have
(70) $c \in X' \wedge$
$\bigvee_{c_i} c' = c_i \wedge$
$\sigma(c\text{'}cond_\pi(c')) \wedge$
$\bigwedge_{i=1..n} \neg(c\text{'}c_i \in X')$

Now, we show that
$c \in X' \wedge$
$\bigvee_{c_i} c' = c_i \wedge$
$\sigma(c\text{'}cond_\pi(c')) \wedge$
$\bigwedge_{i=1..n} \neg(c\text{'}c_i \in X')$
$\Rightarrow$
$c\text{'}c' \in X'$
We assume that
(71) $c \in X' \wedge$
(72) $\bigvee_{c_i} c' = c_i \wedge$
(73) $\sigma(c\text{'}cond_\pi(c')) \wedge$
(74) $\bigwedge_{i=1..n} \neg(c\text{'}c_i \in X')$
We show that
$c\text{'}c' \in X'$
From [7] and [12] on [71], we have
$\exists i \in \{0..n\}$:
(75) $c \in X_i$
From [7] and [12] on [74], we have
$\forall i \in \{0..n\}$:
(76) $\bigwedge_{i=1..n} \neg(c\text{'}c_i \in X_i)$
By Lemma 15.4 on [65], [66], [11], [75], [72], [73] and [76], we have
(77) $c\text{'}c' \in X_i$
From [7] and [12] on [77], we have
$c\text{'}c' \in X'$

Case rule OControl:
Similar to rule IControl using Lemma 15.5.
Case rule TSeq:
We have that
(78) $\mathcal{A} = l_1 \prec l_2 \vee l_2 \prec l_1 \vee l_1 = l_2$
(79) $\pi, \Gamma \vdash exec(l_1)$
(80) $\pi, \Gamma \vdash exec(l_2)$
(81) $\pi, \Gamma \vdash thread(l_1) = thread(l_2)$
(82) $\pi, \Gamma \vdash obj(l_1) = obj(l_2) = \textbf{this} \vee$
$(\neg obj(l_1) = \textbf{this} \wedge \neg obj(l_2) = \textbf{this})$
We show that
$\mathcal{X} \models l_1 \prec l_2 \vee l_2 \prec l_1 \vee l_1 = l_2$
that is
$l_1 \prec_{X'} l_2 \vee l_2 \prec_{X'} l_1 \vee l_1 = l_2$

By the induction hypothesis on [79], [80], [81], [82], and then [2], we have
(83) $\mathcal{X} \models exec(l_1)$
(84) $\mathcal{X} \models exec(l_2)$
(85) $\mathcal{X} \models thread(l_1) = thread(l_2)$
(86) $\mathcal{X} \models obj(l_1) = obj(l_2) = \textbf{this} \vee$
$(\neg obj(l_1) = \textbf{this} \wedge \neg obj(l_2) = \textbf{this})$
that is
(87) $l_1 \in X'$
(88) $l_2 \in X'$
(89) $thread_{X'}(l_1) = thread_{X'}(l_2)$
(90) $obj_{X'}(l_1) = obj_{X'}(l_2) = \textbf{this} \vee$
$(\neg obj_{X'}(l_1) = \textbf{this} \wedge \neg obj_{X'}(l_2) = \textbf{this})$
By [11] and [12] on [87] and [88], we have
$\exists i, j \in 0..n$:
(91) $l_1 \in X_i \wedge (X_i, \sigma) \in [\![p_i]\!]$
(92) $l_2 \in X_j \wedge (X_j, \sigma) \in [\![p_j]\!]$
Case analysis on [90]:
Case
(93) $obj_{X'}(l_1) = obj_{X'}(l_2) = \textbf{this}$
By Lemma 15.8 on [8], [87], [88], [93], we have
$\exists c_1, c_2$:
(94) $l_1 = c_1$
(95) $l_2 = c_2$
By Lemma 15.10 on [91], [92], [94], [95], we have
(96) $thread_X(l_1) = T_i$
(97) $thread_X(l_2) = T_j$
From [96], [97] and [89], we have
(98) $i = j$
By Lemma 15.12 on [91], [92], and [94], [95], and [98], we have
(99) $l_1 \prec_X l_2 \vee l_2 \prec_X l_1 \vee l_1 = l_2$
Case
(100) $\neg obj_{X'}(l_1) = \textbf{this} \wedge$
$\neg obj_{X'}(l_2) = \textbf{this}$
Similar to the previous case where lemmas 15.9, 15.11 and 15.13 are used.

Case rule TLocal:
We have that
(101) $\mathcal{A} = thread(l_1) = thread(l_2)$
(102) $\mathcal{T}(basename(\phi)) = ThreadLocal\ st$
(103) $\pi, \Gamma \vdash exec(l_1) \wedge exec(l_2)$
(104) $\pi, \Gamma \vdash obj(l_1) = obj(l_2) = \phi[u]$
We show that
$\mathcal{X} \models thread(l_1) = thread(l_2)$
that is
$thread_{X'}(l_1) = thread_{X'}(l_2)$
By the induction hypothesis on [104], and then [2], we have
(105) $\mathcal{X} \models exec(l_1) \wedge exec(l_2)$
(106) $\mathcal{X} \models obj(l_1) = obj(l_2) = \phi[u]$
that is
(107) $obj_{X'}(l_1) = obj_{X'}(l_2) = \phi[\sigma(u)]$
(108) $l_1 \in X'$
(109) $l_2 \in X'$
From [107] , we have
(110) $basename(obj_{X'}(l_1)) = \phi$
(111) $index(obj_{X'}(l_1)) = \sigma(u)$
(112) $basename(obj_{X'}(l_2)) = \phi$
(113) $index(obj_X(l_2)) = \sigma(u)$
From Lemma 15.14 on [3], [102], [8], [108] and

[110] we have
(114) $thread_{X'}(l_1) = index(obj_{X'}(l_1))$
From Lemma 15.14 on [3], [102], [8], [109] and [112] we have
(115) $thread_{X'}(l_2) = index(obj_{X'}(l_2))$
From [114] and [111] we have
(116) $thread_{X'}(l_1) = \sigma(u)$
From [115] and [113] we have
(117) $thread_{X'}(l_2) = \sigma(u)$
From [116] and [117] we have
(118) $thread_{X'}(l_1) = thread_{X'}(l_2)$

Case rule Id:
We have that
(119) $\mathcal{A} = obj(\varsigma'c) = \varsigma'\theta \wedge$
$name(\varsigma'c) = n \wedge$
$thread(\varsigma'c) = \varsigma'\tau \wedge$
$arg^*(\varsigma'c) = \varsigma'u^* \wedge$
$retv(\varsigma'c) = \varsigma'x$
(120) $obj_\pi(c) = \theta$
(121) $name_\pi(c) = n$
(122) $thread_\pi(c) = \tau$
(123) $arg_\pi(c) = u$
(124) $retv_\pi(c) = x$
(125) $\pi, \Gamma \vdash exec(\varsigma'c)$
We show that
(126) $\mathcal{X} \models \mathcal{A}$
that is
$obj_{X'}(\varsigma'c) = \sigma(\varsigma'\theta) \wedge$
$name_{X'}(\varsigma'c) = n \wedge$
$thread_{X'}(\varsigma'c) = \sigma(\varsigma'\tau) \wedge$
$arg^*_{X'}(\varsigma'c) = \sigma(\varsigma'u^*) \wedge$
$retv_{X'}(\varsigma'c) = \sigma(\varsigma'x)$
By the induction hypothesis on [125], and then [2], we have
(127) $\mathcal{X} \models exec(\varsigma'c)$
that is
(128) $\varsigma'c \in X'$
From [7] and [128], we have
(129) $\varsigma'c \in X$
From [12] and [129], we have
$\exists i \in \{0..n\}:$
(130) $\varsigma'c \in X_i$
From Lemma 15.1 on [11] and [130], we have
(131) $obj_{X_i}(\varsigma'c) = \varsigma'\theta \wedge$
$name_{X_i}(\varsigma'c) = n \wedge$
$thread_{X_i}(\varsigma'c) = \varsigma'\tau \wedge$
$arg^*_{X_i}(\varsigma'c) = \varsigma'u^* \wedge$
$retv_{X_i}(\varsigma'c) = \varsigma'x$
From [131], [12], we have
(132) $obj_X(\varsigma'c) = \varsigma'\theta \wedge$
$name_X(\varsigma'c) = n \wedge$
$thread_X(\varsigma'c) = \varsigma'\tau \wedge$
$arg^*_X(\varsigma'c) = \varsigma'u^* \wedge$
$retv_X(\varsigma'c) = \varsigma'x$
From [132], [7], we have
(133) $obj_{X'}(\varsigma'c) = \sigma(\varsigma'\theta) \wedge$
$name_{X'}(\varsigma'c) = n \wedge$
$thread_{X'}(\varsigma'c) = \sigma(\varsigma'\tau) \wedge$
$arg^*_{X'}(\varsigma'c) = \sigma(\varsigma'u^*) \wedge$
$retv_{X'}(\varsigma'c) = \sigma(\varsigma'x)$

Case rule Caller:
We have that
(134) $\mathcal{A} =$
$c't = thread(c) \wedge$
$c'x^* = arg^*(c) \wedge$
$\bigvee_{i=1..n}(exec(c'c_i) \wedge arg1(c'c_i) = retv(c))$
(135) $\pi, \Gamma \vdash exec(c)$
(136) $\pi, \Gamma \vdash obj(c) = \textbf{this}$
(137) $\pi, \Gamma \vdash name(c) = n$
(138) $tpar_\pi(n) = t \wedge par1_\pi(n) = x$
(139) $Returns_\pi(n) = \{\overline{c_i}\}$
We show that
$\mathcal{X} \models \mathcal{A}$
that is
$\sigma(c't) = thread_{X'}(c) \wedge$
$\sigma(c'x^*) = arg^*_{X'}(c) \wedge$
$\bigvee_{i=1..n}$
$(c'c_i \in X' \wedge$
$arg1_{X'}(c'c_i) = retv_{X'}(c))$

By induction hypothesis on [135], [136] and [137], and then [2], [6] and [7], we have
(140) $c \in X'$
(141) $obj_{X'}(c) = \textbf{this}$
(142) $name_{X'}(c) = n$
From [7] on [140], [141] and [142], we have
(143) $c \in X$
(144) $obj_X(c) = \textbf{this}$
(145) $name_X(c) = n$
By Lemma 15.15 on [6], [138], [139], [143], [144], and [145], we have
(146) $\sigma(c't) = \sigma(thread_X(c)) \wedge$
(147) $\sigma(c'x^*) = \sigma(arg^*_X(c)) \wedge$
(148) $\bigvee_{i=1..n}$
$(c'c_i \in X \wedge$
$\sigma(arg1_X(c'c_i)) = \sigma(retv_X(c)))$
From [7] on [146], [147], and [148], we have
$\sigma(c't) = thread_{X'}(c) \wedge$
$\sigma(c'x^*) = arg^*_{X'}(c) \wedge$
$\bigvee_{i=1..n}$
$(c'c_i \in X' \wedge$
$arg1_{X'}(c'c_i) = retv_{X'}(c))$

Case rule Ret:

We have that
(149) $tpar_\pi(n) = t \ \wedge \ par1_\pi(n) = x$
(150) $c' \in Returns_\pi(n)$
(151) $\pi, \Gamma \vdash exec(c'c')$
(152) $\mathcal{A} =$
$exec(c) \wedge$
$obj(c) = \textbf{this} \ \wedge \ name(c) = n \wedge$
$thread(c) = c't \ \wedge \ arg^*(c) = c'x^* \wedge$
$retv(c) = arg1(c'c')$
We show that
$\mathcal{X} \models \mathcal{A}$
that is
$c \in X' \wedge$
$obj_{X'}(c) = \textbf{this} \ \wedge \ name_{X'}(c) = n \wedge$
$thread_{X'}(c) = \sigma(c't) \wedge$
$arg^*_{X'}(c) = \sigma(c'x^*) \wedge$
$retv_{X'}(c) = arg1_{X'}(c'c')$

By induction hypothesis on [151],
and then [2], [6] and [7], we have
(153) $c'c' \in X'$
From [7] and [153], we have
(154) $c'c' \in X$
From Lemma 15.17 on [6], [149], [150], and
[154], we have
(155) $c \in X \wedge$
(156) $obj_X(c) = \textbf{this} \ \wedge \ name_X(c) = n \wedge$
(157) $\sigma(thread_X(c)) = \sigma(c't) \wedge$
(158) $\sigma(arg^*_X(c)) = \sigma(c'x^*) \wedge$
(159) $\sigma(retv_X(c)) = \sigma(arg1_X(c'c'))$
From [7] on [155]-[159], we have
$c \in X' \wedge$
$obj_{X'}(c) = \textbf{this} \ \wedge \ name_{X'}(c) = n \wedge$
$thread_{X'}(c) = \sigma(c't) \wedge$
$arg^*_{X'}(c) = \sigma(c'x^*) \wedge$
$retv_{X'}(c) = arg1_{X'}(c'c')$

Case rule Callee:
Similar to rule Ret

Case rule XASym:
We have that
(160) $\pi, \Gamma \vdash l \prec l'$
(161) $\mathcal{A} =$
$\neg(l' \prec l) \ \wedge \ \neg(l' \sim l) \ \wedge \ \neg(l' = l)$
We show that
$\mathcal{X} \models \mathcal{A}$
that is
$\neg(l' \prec_{X'} l) \ \wedge \ \neg(l' \sim_{X'} l) \ \wedge \ \neg(l' = l)$

Straightforward from Lemma 10.1.

Case rule XTotal:

Straightforward from Lemma 10.4.

Case rule X2X:
Straightforward from Lemma 10.5.

Case rule LASym:
We have that
(162) $\pi, \Gamma \vdash l \prec_o l'$
(163) $\mathcal{A} =$
$\neg(l' \prec_o l) \wedge$
$\neg(l' = l)$
We show that
$\mathcal{X} \models \mathcal{A}$
Let
(164) $O = \mathcal{L}(\sigma(o))$
We need to show that
$\neg(l' \prec_O l) \wedge$
$\neg(l' = l)$

Straightforward from Lemma 10.13.

Case rule LTotal:
We have that
(165) $\mathcal{T}_{base}(o) \in LT$
(166) $\pi, \Gamma \vdash exec(l) \ \wedge \ exec(l')$
(167) $\pi, \Gamma \vdash obj(l) = obj(l') = o$
(168) $\mathcal{A} = (l \prec_o l') \ \vee \ (l' \prec_o l) \ \vee \ (l' = l)$
We show that
$\mathcal{X} \models \mathcal{A}$
From [165], let
(169) $O = \mathcal{L}(\sigma(o))$
We need to show that
$(l \prec_O l') \ \vee \ (l' \prec_O l) \ \vee \ (l' = l)$

By induction hypothesis on [166] and [167],
and then [2], [6] and [7], we have
(170) $l \in X'$
(171) $l' \in X'$
(172) $obj_{X'}(l) = \sigma(o)$
(173) $obj_{X'}(l') = \sigma(o)$
From [172] and [173], we have
(174) $l \in X'|\sigma(o)$
(175) $l' \in X'|\sigma(o)$
From [165], we have
(176) $\mathcal{T}_{base}(\sigma(o)) \in LT$
From [10], and [176], we have
(177) $(X'|\sigma(o), \mathcal{L}(\sigma(o))) \in \mathbb{H}_L(o)$
By Lemma 10.15 on [177], [174], [175], we have
$l \prec_O l' \ \vee \ l' \prec_O l \ \vee \ l' = l$

Case rule L2X:
We have that
(178) $\pi, \Gamma \vdash l \prec_o l'$

(179) $\mathcal{A} = exec(l) \ \wedge \ exec(l') \wedge obj(l) = obj(l') = o$

We show that

$\mathcal{X} \models \mathcal{A}$

that is

$l \in X' \ \wedge \ l' \in X' \wedge obj_{X'}(l) = obj_{X'}(l') = \sigma(o)$

Let

(180) $O = \mathcal{L}(\sigma(o))$

By induction hypothesis on [178], and then [2], and [6], we have

(181) $l \prec_O l'$

From [10] on [180], we have

(182) $(X'|\sigma(o), \mathcal{L}(\sigma(o))) \in \mathbb{H}_L(\sigma(o))$

By Lemma 10.16 on [182] and [181], we have

$l \in X' \ \wedge \ l' \in X'$

$obj_{X'}(l) = obj_{X'}(l') = \sigma(o)$

Case rule XTrans:

Straightforward from Lemma 10.2.

Case rule XXTrans:

Straightforward from Lemma 10.3.

Case rule LTrans:

Straightforward from Lemma 10.14.

Case rule TReal:

We have that

(183) $\pi, \Gamma \vdash T \lessdot T'$

(184) $\pi, \Gamma \vdash exec(l) \ \wedge \ thread(l) = T$

(185) $\pi, \Gamma \vdash exec(l') \ \wedge \ thread(l') = T'$

(186) $\mathcal{A} = l \prec l' \ \vee \ l = l'$

We show that

$\mathcal{X} \models \mathcal{A}$

that is

$l \prec_{X'} l'$

By induction hypothesis on [183], [184], and [185], and then [2], [6] and [7], we have

(187) $T \lessdot_{X'} T'$

(188) $l \in X'$

(189) $thread_{X'}(l) = T$

(190) $l' \in X'$

(191) $thread_{X'}(l') = T'$

From [189], we have

(192) $l \in X'|T$

From [189], we have

(193) $l' \in X'|T'$

From [187], we have

(194) $\forall T, T' : X'|T \lhd_H X'|T'$

From [194], [192] and [193], we have

$l \prec_{X'} l'$

Case rule AReg:

We have that

(195) $\mathcal{T}_{base}(reg) = AtomicRegister$

(196) $\pi, \Gamma \vdash isRead_{reg}(l_R)$

(197) $\mathcal{A} = \exists \ell_W : isWriter_{reg}(\ell_W, l_R) \wedge retv(l_R) = arg1(\ell_W)$

Let

(198) $reg' = \sigma(reg)$

(199) $Reg = \mathcal{L}(reg')$

From [195] and [198], we have

(200) $reg' \in AtomicRegister$

From [10] and [200], [199], we have

(201) $(X'|reg', Reg) \in \mathbb{H}_L(reg')$

By the definition of $isWriter$ on [197], we have

(202) $\mathcal{A} = \exists \ell_W : isWrite_{reg}(\ell_W) \wedge \ell_W \prec_{reg} l_R \wedge \forall \ell'_W : isWrite_{reg}(\ell'_W) \Rightarrow (\ell'_W \preceq_{reg} \ell_W \ \vee \ l_R \prec_{reg} \ell'_W) \wedge retv(l_R) = arg1(\ell_W)$

We show that

$\mathcal{X} \models \mathcal{A}$

that is

$\exists l_W : isXWrite_{X',reg'}(l_W) \wedge l_W \prec_{Reg} l_R \wedge \forall l'_W : isXWrite_{X',reg'}(l'_W) \Rightarrow (l'_W \preceq_{Reg} l_W \ \vee \ l_R \preceq_{Reg} l'_W) \wedge retv_{X'}(l_R) = arg1_{X'}(l_W)$

From [196], we have

(203) $\pi, \Gamma \vdash exec(l_R) \wedge obj(l_R) = reg \wedge name(l_R) = read$

By induction hypothesis on [203], and then [2], [6] and [7], we have

(204) $l_R \in X' \wedge obj_{X'}(l_R) = reg' \wedge name_{X'}(l_R) = read$

From the definition of $isXRead$ on [204], we have

(205) $isXRead_{X',reg'}(l_R)$

By Lemma 10.22 on [200], [201] and [205], we have

(206) $\exists l_w : isLWriter_{X'|reg',Reg,reg'}(l_W, l_R) \wedge retv_{X'|reg'}(l_R) = arg1_{X'}(l_W)$

From the definition of $isLWriter$ on [206],

we have
$\exists l_W:$
$isXWrite_{X'|reg',reg'}(l_W) \wedge$
$l_W \prec_{Reg} l_R \wedge$
$\forall l'_W: isXWrite_{X'|reg',reg'}(l'_W) \Rightarrow$
$(l'_W \preceq_{Reg} l_W \vee l_R \preceq_{Reg} l'_W) \wedge$
$retv_{X'|reg'}(l_R) = arg1_{X'|reg'}(l_W)$
After simplification, we have
$\exists l_W:$
$isXWrite_{X',reg'}(l_W) \wedge$
$l_W \prec_{Reg} l_R \wedge$
$\forall l'_W: isXWrite_{X',reg'}(l'_W) \Rightarrow$
$(l'_W \preceq_{Reg} l_W \vee l_R \preceq_{Reg} l'_W) \wedge$
$retv_{X'}(l_R) = arg1_{X'}(l_W)$

Case rule BReg:
Similar to rule AReg by Lemma 10.21.

Case rule CASRegRead:
By Lemma 10.24.

Case rule CASRegCAST:
By Lemma 10.25.

Case rule CASRegCASF:
By Lemma 10.25.

Case rule Lock:
We have that
(207) $\mathcal{T}_{base}(lo) = Lock$
(208) $\pi, \Gamma \vdash isOwnerRespecting(lo)$
(209) $\pi, \Gamma \vdash isLock_{lo}(l_{l_1})$
(210) $\pi, \Gamma \vdash isUnlock_{lo}(l_{u_2})$
(211) $\pi, \Gamma \vdash l_{l_1} \prec_{lo} l_{u_2}$
(212) $\mathcal{A} = \exists \ell_{u_1}, \ell_{l_2}:$
$isUnlock_{lo}(\ell_{u_1}) \wedge$
$thread(\ell_{u_1}) = thread(l_{l_1}) \wedge$
$isLock_{lo}(\ell_{l_2}) \wedge$
$thread(l_{u_2}) = thread(\ell_{l_2}) \wedge$
$\ell_{u_1} \prec_{lo} \ell_{l_2}$
Let
(213) $lo' = \sigma(lo)$
(214) $L = \mathcal{L}(lo')$
We show that
$\mathcal{X} \models \mathcal{A}$
that is
(215) $\exists l_{u_1}, l_{l_2}:$
$isXUnlock_{X',lo'}(l_{u_1}) \wedge$
$thread_{X'}(l_{l_1}) = thread_{X'}(l_{u_1}) \wedge$
$isXLock_{X',lo'}(l_{l_2}) \wedge$
$thread_{X'}(l_{l_2}) = thread_{X'}(l_{u_2}) \wedge$
$l_{u_1} \prec_L l_{l_2}$
By induction hypothesis on [208]-[211], and then [2], [6] and [7], we have
(216) $isXOwnerRespecting_{lo'}(X') \wedge$
(217) $isXLock_{X',lo'}(l_{l_1}) \wedge$
(218) $isXUnlock_{X',lo'}(l_{u_2}) \wedge$
(219) $l_{l_1} \prec_L l_{u_2}$)
From [216]-[219], we have
(220) $isXOwnerRespecting_{lo'}(X'|lo') \wedge$
(221) $isXLock_{X'|lo',lo'}(l_{l_1}) \wedge$
(222) $isXUnlock_{X'|lo',lo'}(l_{u_2}) \wedge$
(223) $l_{l_1} \prec_L l_{u_2}$)
From [207] and [213], we have
(224) $lo' \in Lock$
From Lemma 10.29 on [224], and [220]-[223], we have
$\exists l_{u_1}, l_{l_2}:$
(225) $isXUnlock_{X'|lo',lo'}(l_{u_1}) \wedge$
(226) $thread_{X'|lo'}(l_{l_1}) = thread_{X'|lo'}(l_{u_1}) \wedge$
(227) $isXLock_{X'|lo',lo'}(l_{l_2}) \wedge$
(228) $thread_{X'|lo'}(l_{l_2}) = thread_{X'|lo'}(l_{u_2}) \wedge$
(229) $l_{u_1} \prec_L l_{l_2}$
From [225]-[229], we have
$\exists l_{u_1}, l_{l_2}:$
(230) $isXUnlock_{X',lo'}(l_{u_1}) \wedge$
(231) $thread_{X'}(l_{l_1}) = thread_{X'}(l_{u_1}) \wedge$
(232) $isXLock_{X',lo'}(l_{l_2}) \wedge$
(233) $thread_{X'}(l_{l_2}) = thread_{X'}(l_{u_2}) \wedge$
(234) $l_{u_1} \prec_L l_{l_2}$

Case rule LockReadL:
Similar to the proof of rule Lock using Lemma 10.30.

Case rule LockReadR:
Similar to the proof of rule Lock using Lemma 10.31.

Case rule TryLock:
Similar to the proof of rule Lock using Lemma 10.35.

Case rule TryLockReadL:
Similar to the proof of rule Lock using Lemma 10.36.

Case rule TryLockReadR:
Similar to the proof of rule Lock using Lemma 10.37.

Case rule SCounter:
By Lemma 10.41.

Case rule BasicSetContains:
By Lemma 10.43.

Case rule BasicSetAdd:
By Lemma 10.44.

Case rule BasicMapGet:
By Lemma 10.46.

Case rule BasicMapPut:
By Lemma 10.47.

The basic inference rules and the equivalence and arithmetic rules are standard. □

Lemma 15.1.
$\forall p, X, \sigma, \varsigma, c' :$
$((X, \sigma) \in [\![p]\!] \ \wedge \ \varsigma\text{'}c' \in X)$
$\Rightarrow$
$(obj_X(\varsigma\text{'}c') = \varsigma\text{'}obj_\pi(c') \ \wedge \ thread_X(\varsigma\text{'}c') = \varsigma\text{'}thread_\pi(c') \ \wedge$
$name_X(\varsigma\text{'}c') = name_\pi(c_i) \ \wedge \ arg1_X(\varsigma\text{'}c') = \varsigma\text{'}arg1_\pi(c') \ \wedge$
$retv_X(\varsigma\text{'}c') = \varsigma\text{'}retv_\pi(c'))$.

Proof.

Structural induction on p:
(1) Case $p = c \triangleright \ n_\tau(u^*){:}x$
Straightforward form definition [1].
(2) Case $p = p_1;\ p_2$
Straightforward form definition [11] and the induction hypothesis.
(3) Case $p =$ **if** $b\ p_1$ **else** p_2
Straightforward form definition [12] and the induction hypothesis.
□

Lemma 15.2.
$\forall p, X, \sigma, \varsigma, c' :$
$((X, \sigma) \in [\![p]\!] \ \wedge \ \varsigma\text{'}c' \in X)$
$\Rightarrow$
$basename(obj_X(\varsigma\text{'}c')) = obj_\pi(c') \ \wedge \ name_X(\varsigma\text{'}c') = name_\pi(c')$.

Proof.

Structural induction on p:
(1) Case $p = c \triangleright \ n_\tau(u^*){:}x$
Straightforward form definition [1] and
$basename(c\text{'}obj_\pi(c')) = basename(obj_\pi(c'))$.
(2) Case $p = p_1;\ p_2$
Straightforward form definition [11] and the induction hypothesis.
(3) Case $p =$ **if** $b\ p_1$ **else** p_2
Straightforward form definition [12] and the induction hypothesis. □

Lemma 15.3.
Let
$Labels(name_\pi(c)) = \{\overline{c_i}\}$
$PreReturns_\pi(c') = \{\overline{c_r}\}$
$\forall p, X, \sigma, c, c' :$
$((X, \sigma) \in [\![p]\!] \ \wedge \ c\text{'}c' \in X)$
$\Rightarrow$
$c \in X \ \wedge \ \bigvee_{c_i} c' = c_i\sigma(c\text{'}cond_\pi(c')) \ \wedge \ \bigwedge_{c_r} \neg(c\text{'}c_r \in X)$.

Proof.

Structural induction on p:
(1) Case $p = c \triangleright \ n_\tau(u^*){:}x$
Straightforward form definition [1]
(2) Case $p = p_1;\ p_2$
Straightforward form definition [11], the induction hypothesis and the uniqueness of label c.
(3) Case $p =$ **if** $b\ p_1$ **else** p_2
Straightforward form definition [12] and the induction hypothesis. □

Lemma 15.4.
Let

$Labels(name_\pi(c)) = \{\overline{c_i}\}$
$PreReturns_\pi(c') = \{\overline{c_r}\}$
$\forall p, X, \sigma, c, c' :$
$((X,\sigma) \in [\![p]\!] \ \wedge$
$c \in X \ \wedge$
$\bigvee_{c_i} c' = c_i \ \wedge$
$\sigma(c\text{'}cond_\pi(c')) \ \wedge$
$\bigwedge_{c_r} \neg(c\text{'}c_r \in X))$
$\Rightarrow$
$c\text{'}c' \in X.$

Proof.

Structural induction on p:
(1) Case $p = c \triangleright \ n_\tau(u^*){:}x$
Straightforward form definition [1]
(2) Case $p = p_1;\ p_2$
Straightforward form definition [11], and the induction hypothesis.
(3) Case $p =$ **if** $b\ p_1$ **else** p_2
Straightforward form definition [12] and the induction hypothesis. □

Lemma 15.5.
Let
$\forall p, X, \sigma, c :$
$(X,\sigma) \in [\![p]\!]$
$\Rightarrow$
$\sigma(cond_\pi(c))$
$\Leftrightarrow$
$c \in X.$

Proof.

Structural induction on p:
(1) Case $p = c \triangleright \ n_\tau(u^*){:}x$
Straightforward form definition [1]
$cond_\pi(c) = true$
(2) Case $p = p_1;\ p_2$
Straightforward form definition [11], and the induction hypothesis.
(3) Case $p =$ **if** $b\ p_1$ **else** p_2
Straightforward form definition [12] and the induction hypothesis.
$\sigma(b)$ for the then part and
$\neg\sigma(b)$ for the else part. □

Lemma 15.6.
$\forall \pi, X, \varsigma, c_1, c_2 :$
$X \in \mathbb{H}(\pi) \ \wedge$
$\varsigma\text{'}c_1 \in X \ \wedge$
$\varsigma\text{'}c_2 \in X \ \wedge$
$c_1 \rightarrow_\pi c_2$
$\Rightarrow$
$\varsigma\text{'}c_1 \prec_X \varsigma\text{'}c_2.$

Proof.

Case analysis on $c_1 \rightarrow_\pi c_2$
(1) Case: the initialization order
Straightforward form definition [17] and [13].
$X = X_0 \cdot X'$
(2) Case: the sequential order of the sequential programs p_i
Straightforward form structural induction on p_i and definition [1], [11], and [12].
$X = X_1 \cdot X_2$
(3) Case: $\rightarrow_n$ of a method n.
Straightforward form definition [1]
$\forall c_i, c_j \in \{\overline{c_i}\}:$
$\Big((c_i \rightarrow_n c_j) \ \wedge \ c\text{'}c_i \in X' \ \wedge \ c\text{'}c_j \in X'\Big) \Rightarrow$
$c\text{'}c_i \prec_{X'} c\text{'}c_j$ □

Lemma 15.7.
$\forall \pi, Xc, c' :$
$X \in \mathbb{H}(\pi) \ \wedge \ c'c' \in X) \Rightarrow$
$(iEv(c) \lhd_X iEv(c'c') \ \wedge \ rEv(c'c') \lhd_X rEv(c)).$

Proof.

We have that
(1) $X \in \mathbb{H}(\pi)$
(2) $c'c' \in X$
We show that
$iEv(c) \lhd_X iEv(c'c')$
$rEv(c'c') \lhd_X rEv(c)$

From definition 17 and [13]on [1] and [2], we have
$\exists X_i :$
(3) $(X_i, \sigma) \in [\![p_i]\!]$
(4) $c'c' \in X_i$
(5) $X_i \Subset X$
We show that
(6) $iEv(c) \lhd_{X_i} iEv(c'c')$
(7) $rEv(c'c') \lhd_{X_i} rEv(c)$
Structural induction on p:
(8) Case $p = c \triangleright \ n_\tau(u^*)$:$x$
Straightforward form definition [1]
$X = inv(c \triangleright \ n_\tau(u)) \cdot X' \cdot ret(c \triangleright \ x')$
(9) Case $p = p_1; \ p_2$
Straightforward form definition [11],
the induction hypothesis and
the uniqueness of label c.
(10) Case $p =$ **if** $b \ p_1$ **else** p_2
Straightforward form definition [12] and
the induction hypothesis.
From [5] on [6] and [7], we have
$iEv(c) \lhd_X iEv(c'c')$
$rEv(c'c') \lhd_X rEv(c)$
□

Lemma 15.8.
$\forall \pi, X, \sigma, c :$
$X \in \mathbb{H}(\pi) \ \wedge$
$l \in X \ \wedge$
$obj_X(l) = \textbf{\textit{this}} \ \wedge$
$\Rightarrow$
$\exists c : l = c.$

Proof.

From definition 17 and [13], we have
$\exists X_i :$
(1) $(X_i, \sigma) \in [\![p_i]\!]$
(2) $l \in X_i$
(3) $X_i \Subset X$
Straightforward form structural induction on p_i □

Lemma 15.9.
$\forall \pi, X, \sigma, c :$
$X \in \mathbb{H}(\pi) \ \wedge$
$l \in X \ \wedge$
$\neg obj_X(l) = \textbf{\textit{this}} \ \wedge$
$\Rightarrow$
$\exists c, c' : l = c'c'.$

Proof. Similar to Lemma 15.8. □

Lemma 15.10.
$\forall \pi, \mathcal{T}, \mathcal{D}, \mathcal{P}, p_0, ..., p_n, X, \sigma, c :$
$(\pi = (\mathcal{T}, \mathcal{D}, \mathcal{P}) \ \wedge$
$\mathcal{P} = p_0, (p_1 || p_2 || ... || p_n) \ \wedge$
$(X, \sigma) \in [\![p_i]\!] \ \wedge$
$c \in X \ \wedge$
$\Rightarrow$
$thread_X(c) = i.$

Proof.

By structural induction on p_i, we have
(1) $c \in Labels(p_i)$
(2) $thread_X(c) = thread_\pi(c)$
From the well-formedness conditions, we have
The thread argument of each method call is the identifier of the thread in which it is called.
(3) $\forall c \in Labels(p_i) : thread_\pi(c) = i$
From [1], [2] and [3], we have
(4) $thread_X(c) = T_i$ □

Lemma 15.11.
$\forall \pi, \mathcal{T}, \mathcal{D}, \mathcal{P}, p_0, ..., p_n, X, \sigma, c :$
$(\pi = (\mathcal{T}, \mathcal{D}, \mathcal{P}) \ \wedge$
$\mathcal{P} = p_0, (p_1 || p_2 || ... || p_n) \ \wedge$
$(X, \sigma) \in [\![p_i]\!] \ \wedge$
$c'c' \in X \ \wedge$
$\Rightarrow$
$\sigma(thread_X(c'c')) = i.$

Proof.

By structural induction on p_i, we have
$\exists n, \tau :$
(1) $c' \in Labels(n)$
(2) $thread_X(c'c') = c' thread_\pi(c')$
(3) $\sigma(c' tpar_\pi(n)) = \sigma(\tau)$
(4) $c \in X$
(5) $thread_X(c) = \tau$
By Lemma 15.10 on [4], we have
(6) $thread_X(c) = i$
From the well-formedness conditions, we have
The thread argument of each method call is the identifier of the thread in which it is called.
(7) $\forall c' \in Labels(n) : thread_\pi(c') = tpar_\pi(n)$
From [2], [7], [3], [5], and [6], we have
(8) $\sigma(thread_X(c'c')) = i$ □

Lemma 15.12.
$\forall p, X, \sigma, c_1, c_2 :$
$(X, \sigma) \in [\![p]\!] \ \wedge$
$c_1 \in X \ \wedge$
$c_2 \in X \ \wedge$
$\Rightarrow$
$c_1 \prec_X c_2 \ \vee \ c_2 \prec_X c_1 \ \vee \ c_1 = c_2.$

Proof. Straightforward structural induction on p. □

Lemma 15.13.
$\forall p, X, \sigma, c_1, c_2, c_3, c_4 :$
$(X, \sigma) \in [\![p]\!] \ \wedge$
$c_1'c_2 \in X \ \wedge$
$c_3'c_4 \in X \ \wedge$
$\Rightarrow$
$c_1'c_2 \prec_X c_3'c_4 \ \vee \ c_3'c_4 \prec_X c_1'c_2 \ \vee \ c_1'c_2 = c_3'c_4.$

Proof. Straightforward structural induction on p. □

Lemma 15.14.
$\forall \pi, X, \phi, st :$
$\pi = (\mathcal{T}, \mathcal{D}, \mathcal{P}) \ \wedge$
$\mathcal{T}(\phi) = Threadlocal\ st \ \wedge$
$X \in \mathbb{H}(\pi) \ \wedge$
$l \in X \ \wedge$
$basename(obj_X(l)) = \phi$
$\Rightarrow$
$thread_X(l)) = index(obj_X(l)).$

Proof.

We have
(1) $\pi = (\mathcal{T}, \mathcal{D}, \mathcal{P})$
(2) $\mathcal{T}(\phi) = Threadlocal\ st$
(3) $X \in \mathbb{H}(\pi)$
(4) $l \in X$
(5) $basename(obj_X(l)) = \phi$
From definition 17 and 13 on [3] and [5], we have
$\exists X_i$:
(6) $l \in X_i$
(7) $(X_i, \sigma) \in [\![p_i]\!]$
(8) $basename(obj_{X_i}(l)) = \phi$
(9) $X_i \Subset X$
We show that
(10) $thread_{X_i}(l) = index(obj_{X_i}(l))$
Structural induction on p_i:
(11) Case $p_i = c \triangleright\ n_\tau(u^*):x$
Form definition [1], we have
(12) $l = c\text{'}c'$
(13) $index(object_{X_i}(c\text{'}c')) = c\text{'}index_\pi(c')$
(14) $thread_{X_i}(c\text{'}c') = c\text{'}thread_\pi(c')$
From the well-formedness conditions, we have
The thread argument of each method call is the identifier of the thread in which it is called.
(15) $\forall c' \in Labels(n) : thread_\pi(c') = tpar_\pi(n)$
From the well-formedness conditions, we have
The array access index to every thread-local object is the current thread identifier.
(16) $\forall \phi, st, c' :$
$\mathcal{T}(\phi) = Threadlocal\ st \wedge$
$c' \in Labels(n) \Rightarrow$
$index_\pi(c') = tpar_\pi(n)$
From [13], [14], [15], [16], we have
(17) $thread_{X_i}(l) = index(obj_{X_i}(l))$
(18) Case $p_i = p'\ p''$
Straightforward form definition [11], the induction hypothesis and the uniqueness of label l.
(19) Case $p = \textbf{if}\ b\ p_1\ \textbf{else}\ p_2$
Straightforward form definition [12] and the induction hypothesis.
From [10] and [9], we have
$thread_X(l) = index(obj_X(l))$
□

Lemma 15.15.
$\forall \pi, X, \sigma, \mathcal{L}, c, n, t, x :$
$(X, \sigma, \mathcal{L}) \in [\![\pi]\!]$
$tpar_\pi(n) = t\ \wedge\ par1_\pi(n) = x$
$Returns_\pi(n) = \{\overline{c_i}\}$
$c \in X$
$obj_X(c) = \textbf{\textit{this}}$
$name_X(c) = n$
$\Rightarrow$
$\sigma(c\text{'}t) = thread_X(c)\ \wedge$
$\sigma(c\text{'}x^*) = arg^*_X(c)\ \wedge$
$\bigvee_{i=1..n}$
$(c\text{'}c_i \in X\ \wedge$
$arg1_X(c\text{'}c_i) = retv_X(c))$.

Proof.

We have that
(1) $(X, \sigma, \mathcal{L}) \in [\![\pi]\!]$
(2) $tpar_\pi(n) = t\ \wedge\ par1\pi(n) = x$
(3) $Returns_\pi(n) = \{\overline{c_i}\}$
(4) $c \in X$
(5) $obj_X(c) = \textbf{this}$
(6) $name_X(c) = n$
We show that
$\sigma(c\text{'}t) = \sigma(thread_X(c)) \wedge$
$\sigma(c\text{'}x^*) = \sigma(arg^*_X(c)) \wedge$
$\bigvee_{i=1..n}$
$(c\text{'}c_i \in X \wedge$
$\sigma(arg1_X(c\text{'}c_i)) = \sigma(retv_X(c)))$
From definition 13 on [1], [4], [5], and [6], we have
$\exists X_i$:
(7) $(X_i, \sigma) \in [\![p_i]\!]$
(8) $c \in X_i$
(9) $obj_{X_i}(c) = \textbf{this}$
(10) $name_{X_i}(c) = n$
(11) $X_i \Subset X$
We show that
(12) $\sigma(c\text{'}t) = \sigma(thread_{X_i}(c)) \wedge$
(13) $\sigma(c\text{'}x^*) = \sigma(arg^*_{X_i}(c)) \wedge$
(14) $\bigvee_{i=1..n}$
$(c\text{'}c_i \in X_i \wedge$

$\sigma(arg1_{X_i}(c'c_i)) = \sigma(retv_{X_i}(c)))$
Structural induction on p_i:
(15) Case $p_i = c \triangleright\ n_\tau(u^*):x$
From the Well-formedness
condition of specifications that
Every branch of every method definition ends
in a return statement.
we have
$\exists c_r \in \{\overline{c_r}\}: \sigma(c'cond_\pi(c_i))$
The rest is straightforward form the following
conditions of definition [1]
$\forall c_i \in \{\overline{c_i}\}:$
$c'c_i \in X' \Leftrightarrow$
$(\sigma(c'cond_\pi(c_i)) \wedge$
$\forall c_j \in PreReturns_\pi(c_i) \Rightarrow \neg c'c_j \in X')$
and
$\forall c_r \in \{\overline{c_r}\}:$
$c'c_r \in X' \Rightarrow \sigma(x') = \sigma(c'arg1_\pi(c_r))$
(16) Case $p_i = p'\ p''$
Straightforward form definition [11],
the induction hypothesis and
the uniqueness of label c.
(17) Case $p =$ **if** $b\ p_1$ **else** p_2
Straightforward form definition [12] and
the induction hypothesis.
From [11] on [12], [13] and [14], we have
$\sigma(c't) = \sigma(thread_X(c)) \wedge$
$\sigma(c'x^*) = \sigma(arg^*_X(c)) \wedge$
$\bigvee_{i=1..n}$
$(c'c_i \in X \wedge$
$\sigma(arg1_X(c'c_i)) = \sigma(retv_X(c)))$ □

Lemma 15.16.
$\forall X, \sigma, c, n, \tau, u, x':$
$(X, \sigma) \in [\![c \triangleright\ n_\tau(u):x]\!]$
$c', c'' \in Returns_\pi(n)$
$c'c' \in X\ \wedge\ c'c'' \in X$
$\Rightarrow$
$c' = c''.$

Proof.

We have that
(1) $(X, \sigma) \in [\![c \triangleright\ n_\tau(u):x]\!]$
(2) $c' \in Returns_\pi(n)$
(3) $c'' \in Returns_\pi(n)$
(4) $c'c' \in X$
(5) $c'c'' \in X$
We show that
$c' = c''$

We consider three cases

Case
$c' = c''$
Obvious
Case
$c' \in PreReturns_\pi(c'')$
By definition [1] on [5], we have
$\neg c'c' \in X$
which is contradiction to [4].
Case
$c'' \in PreReturns_\pi(c')$
By definition [1] on [4], we have
$\neg c'c'' \in X$
which is contradiction to [5]. □

Lemma 15.17.
$\forall \pi, X, \sigma, \mathcal{L}, c, c', n, t, x:$
$(X, \sigma, \mathcal{L}) \in [\![\pi]\!]$
$tpar_\pi(n) = t\ \wedge\ par1_\pi(n) = x$
$c' \in Returns_\pi(n)$
$c'c' \in X$
$\Rightarrow$
$c \in X\ \wedge$
$obj_X(c) = \textbf{\textit{this}} \wedge\ name_X(c) = n\ \wedge$
$\sigma(thread_X(c)) = \sigma(c't)\ \wedge$
$\sigma(arg^*_X(c)) = \sigma(c'x^*)\ \wedge$
$\sigma(retv_X(c)) = \sigma(arg1_X(c'c')).$

Proof.

We have that
(1) $(X, \sigma, \mathcal{L}) \in [\![\pi]\!]$
(2) $tpar_\pi(n) = t \ \wedge \ par1\pi(n) = x$
(3) $c' \in Returns_\pi(n)$
(4) c'$c' \in X$
We show that
$c \in X \wedge$
$obj_X(c) = \textbf{this} \ \wedge \ name_X(c) = n \wedge$
$\sigma(thread_X(c)) = \sigma(c\text{'}t) \wedge$
$\sigma(arg^*_X(c)) = \sigma(c\text{'}x^*) \wedge$
$\sigma(retv_X(c)) = \sigma(arg1_X(c\text{'}c'))$

From definition 13 on [1] and [4], we have
$\exists X_i$:
(5) $(X_i, \sigma) \in [\![p_i]\!]$
(6) c'$c' \in X_i$
(7) $X_i \Subset X$
We show that
(8) $c \in X_i \wedge$
(9) $obj_{X_i}(c) = \textbf{this} \ \wedge \ name_{X_i}(c) = n \wedge$
(10) $\sigma(thread_{X_i}(c)) = \sigma(c\text{'}t) \wedge$
(11) $\sigma(arg^*_{X_i}(c)) = \sigma(c\text{'}x^*) \wedge$
(12) $\sigma(retv_{X_i}(c)) = \sigma(arg1_{X_i}(c\text{'}c'))$
Structural induction on p_i:
(13) Case $p_i = c \triangleright n_\tau(u^*)$:$x$
Straightforward form definition [1] and Lemma 15.16.
(14) Case $p_i = p' \ p''$
Straightforward form definition [11], the induction hypothesis and the uniqueness of label c.
(15) Case $p = \textbf{if} \ b \ p_1 \ \textbf{else} \ p_2$
Straightforward form definition [12] and the induction hypothesis.
From [11] on [8]-[12], we have
$c \in X \wedge$
$obj_X(c) = \textbf{this} \ \wedge \ name_X(c) = n \wedge$
$\sigma(thread_X(c)) = \sigma(c\text{'}t) \wedge$
$\sigma(arg^*_X(c)) = \sigma(c\text{'}x^*) \wedge$
$\sigma(retv_X(c)) = \sigma(arg1_X(c\text{'}c'))$
□

15.3 Derived Rules

P2L:
Derived from rule P2X and rule X2L.

IX2OX:
Derived from rule X2X, rule Callee, rule TSeq, rule OX2IX, and rule XASym.

XLTrans:
Derived from rule L2X, rule XTotal, rule XTrans, rule XXTrans, rule X2L, and rule LASym.

X2L':
Derived from rule L2X, rule XTotal, rule X2L, and rule LASym.

AReg':
Derived from rule AReg and the following

$$(\pi, \Gamma \vdash isWriter_{reg}(l_W, l_R) \ \wedge \ isWriter_{reg}(l_{W'}, l_R)) \Rightarrow (\pi, \Gamma \vdash l_W = l_{W'})$$

BReg':
Derived from rule BReg and the following

$$\pi, \Gamma \vdash isSequential(reg) \Rightarrow \pi, \Gamma \vdash \forall \ell : (isRead_{reg}(\ell) \ \vee \ isWrite_{reg}(\ell)) \Rightarrow isRaceFree_{reg}(\ell)$$

TReg:
Derived from rule TLocal, rule TSeq and rule BReg'.

CASRegRead':
Derived from rule CASRegRead and the following

$$(\pi, \Gamma \vdash isCWriter_{reg}(l_W, l_R) \ \wedge \ isCWriter_{reg}(l_{W'}, l_R)) \Rightarrow (\pi, \Gamma \vdash l_W = l_{W'})$$

SCounter':
Derived from rule LTotal and rule SCounter.

BasicMapGet':
Derived from rule BasicMapGet.

BasicMapPut':
Derived from rule BasicMapPut.

DisjSyllL:
Derived form rule DisjElim and rule NegElim.

DisjSyllR:
Derived form rule DisjElim and rule NegElim.

CondElim':
Derived form rule Premise, rule CondElim, and rule NegIntro.

Other Lemmas:
Lemma 13.1:
Derived from rule Premise.

Lemma 13.2:
Derived from rule Premise.

15.4 Client Assertions

Let us define

$$Inits(X) = \{l \mid l \in X \ \wedge \ obj_X(l) = this \ \wedge \ name_X(l) = init\} \quad (144)$$

$$Reads(X) = \{l \mid l \in X \ \wedge \ obj_X(l) = this \ \wedge \ name_X(l) = read\} \quad (145)$$

$$Writes(X) = \{l \mid l \in X \ \wedge \ obj_X(l) = this \ \wedge \ name_X(l) = write\} \quad (146)$$

$$Commits(X) = \{l \mid l \in X \ \wedge \ obj_X(l) = this \ \wedge \ name_X(l) = commit\} \quad (147)$$

$$Committed(X) = \{T \mid \exists l : l \in Commits(X) \ \wedge \ thread_X(l) = T \ \wedge \ retv_X(l) = \mathbb{C}\} \quad (148)$$

$$Aborted(X) = \{T \mid \exists l : l \in X \ \wedge \ obj_X(l) = this \ \wedge \ thread_X(l) = T \ \wedge \ retv_X(l) = \mathbb{A}\} \quad (149)$$

Lemma 15.18.
$\forall X, \sigma, c:$
$(X, \sigma) \in [\![trans_j]\!] \ \wedge$
$c \in X$
$\Rightarrow$
$(c \in Inits(X) \ \wedge \ c = IL_j \ \vee$
$c \in Reads(X) \ \vee$
$c \in Writes(X) \ \vee$
$c \in Commits(X) \ \wedge \ c = CL_j) \ \wedge$
$(IL_j \preceq c) \ \wedge$
$(CL_j \in X \Rightarrow c \preceq CL_j)$

Proof.
Case $j = 0$:

Case $0 < j \leq n$:
Derived from Equation 82, induction on the structure of *op* and Equation 12. □

Lemma 15.19.
$\forall X, \sigma:$
$(X, \sigma) \in [\![trans_j]\!]$
$\Rightarrow$
$\exists c:$
$c \in X \ \wedge \ obj_X(c) = this \ \wedge \ thread_X(c) = j \ \wedge$
$(retv_X(c) = \mathbb{C} \ \vee \ retv_X(c) = \mathbb{A})$

Proof.
Case $j = 0$:
Derived from Equation 81, Equation 11, Equation 1 and the well-formedness condition
$\forall c' \in Returns_\pi(commit) : retv_\pi(c') = \mathbb{C} \ \vee \ retv_\pi(c') = \mathbb{A}$.
Case $0 < j \leq n$:
Derived from Equation 82, induction on the structure of *op* and Equation 12, Equation 1 and the well-formedness condition
$\forall c' \in Returns_\pi(commit) : retv_\pi(c') = \mathbb{C} \ \vee \ retv_\pi(c') = \mathbb{A}$. □

Lemma 15.20.
$\forall X, \sigma, c, c':$
$(X, \sigma) \in [\![trans_j]\!]$
$c \in X \ \wedge \ obj_X(c) = this \ \wedge \ thread_X(c) = j \ \wedge$
$c' \in X \ \wedge \ obj_X(c') = this \ \wedge \ thread_X(c') = j \ \wedge$
$(retv_X(c) = \mathbb{C} \ \vee \ retv_X(c) = \mathbb{C}) \ \vee \ (retv_X(c') = \mathbb{A} \ \vee \ retv_X(c') = \mathbb{A}) \Rightarrow$
$c = c'$

Proof.
Case $j = 0$:
Derived from Equation 81, Equation 11, Equation 1 and the well-formedness conditions
$\forall c \in Returns_\pi(init) : arg1_\pi(c) = ok$

$\forall c \in Returns_\pi(write) : arg1_\pi(c) \neq \mathbb{C}$
and that in every execution of the transaction $trans_0$, all the *write* method calls return *ok*.
Case $0 < j \leq n$:
Derived from Equation 82, induction on the structure of *op* and Equation 12, Equation 1 and the following well-formedness conditions
$\forall c \in Returns_\pi(init) : arg1_\pi(c) = ok$
$\forall c \in Returns_\pi(read) : arg1_\pi(c) \neq \mathbb{C}$
$\forall c \in Returns_\pi(write) : arg1_\pi(c) \neq \mathbb{C}$
$\forall c \in Returns_\pi(commit) : arg1_\pi(c) = \mathbb{C} \ \vee \ arg1_\pi(c) = \mathbb{A}$ □

Lemma 15.21.
$\forall \pi \in \Pi_{TM} : \forall X \in \mathbb{H}(\pi) : \forall T \in Trans(X) : Let\ l = commitOf(T) : l \in Inits(X) \ \wedge \ thread_X(l) = T$

Proof. Derived from Equation 81, Equation 82, Equation 83, Equation 17, Equation 13, and Equation 11. □

Lemma 15.22.
$\forall \pi \in \Pi_{TM} : \forall X \in \mathbb{H}(\pi) : \forall l, l' :$
$(l \in Inits(X) \ \wedge \ l' \in Inits(X) \ \wedge \ thread_X(l) = thread_X(l')) \Rightarrow$
$l = l'$

Proof. Derived from Equation 17, Equation 13, Lemma 15.8, Lemma 15.10, and Lemma 15.18. □

Lemma 15.23.
$\forall \pi \in \Pi_{TM} : \forall X \in \mathbb{H}(\pi) : \forall l, l' :$
$(l \in Inits(X) \ \wedge \ l' \in X \ \wedge \ obj_X(l') = this \ \wedge \ thread_X(l) = thread_X(l')) \Rightarrow$
$l \preceq_X l'$

Proof. Derived from Equation 17, Equation 13, Lemma 15.8, Lemma 15.10, and Lemma 15.18. □

Lemma 15.24.
$\forall \pi \in \Pi_{TM} : \forall X \in \mathbb{H}(\pi) : \forall T \in Trans(X)$
$Let\ l = commitOf(T) :$
$T \in Committed(X) \Rightarrow$
$(l \in Commits(X) \ \wedge \ thread_X(l) = T)$

Proof. Derived from Equation 84, Equation 17, Equation 13, Lemma 15.8, Lemma 15.18 and Lemma 15.10. □

Lemma 15.25.
$\forall \pi \in \Pi_{TM} : \forall X \in \mathbb{H}(\pi) : \forall l, l' :$
$(l \in Commits(X) \ \wedge \ l' \in Commits(X) \ \wedge \ thread_X(l) = thread_X(l')) \Rightarrow$
$l = l'$

Proof. Derived from Equation 17, Equation 13, Lemma 15.8, Lemma 15.10 and Lemma 15.18. □

Lemma 15.26.
$\forall \pi \in \Pi_{TM} : \forall X \in \mathbb{H}(\pi) : \forall l, l' :$
$(l \in X \ \wedge \ obj_X(l) = this \ \wedge \ l' \in Commits(X) \ \wedge \ thread_X(l) = thread_X(l')) \Rightarrow$
$l \preceq_X l'$

Proof. Derived from Equation 17, Equation 13, Lemma 15.8, Lemma 15.10 and Lemma 15.18. □

Lemma 15.27.
$\forall \pi \in \Pi_{TM} : \forall X \in \mathbb{H}(\pi) : \forall t : 0 \leq t \leq n$
$(t \in Committed(X) \ \wedge \ t \in \neg Aborted(X)) \ \vee \ (t \in Aborted(X) \ \wedge \ t \in \neg Committed(X))$

Proof. Derived from Equation 17, Equation 13, Lemma 15.19, and Lemma 15.20. □

Lemma 14.1
$\forall \pi \in \Pi_{TM} : \pi \models \Gamma_0$.

Proof. Derived from Equations 144-149, Equations 136-142, the definition of $\models$ (Figure 8), Definition 13.3 and Lemmas 15.21-15.27. □