Appendix

Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Simple Example</td>
<td>16</td>
</tr>
<tr>
<td>9</td>
<td>Algorithm Description</td>
<td>20</td>
</tr>
<tr>
<td>10</td>
<td>Semantics</td>
<td>22</td>
</tr>
<tr>
<td>10.1</td>
<td>Execution Histories</td>
<td>23</td>
</tr>
<tr>
<td>10.2</td>
<td>Synchronization Object Types</td>
<td>24</td>
</tr>
<tr>
<td>10.3</td>
<td>Transaction Histories</td>
<td>34</td>
</tr>
<tr>
<td>11</td>
<td>Inference Rules</td>
<td>35</td>
</tr>
<tr>
<td>12</td>
<td>Dekker Algorithm</td>
<td>49</td>
</tr>
<tr>
<td>13</td>
<td>Soundness</td>
<td>53</td>
</tr>
<tr>
<td>14</td>
<td>Client assertions</td>
<td>54</td>
</tr>
<tr>
<td>15</td>
<td>Proofs</td>
<td>55</td>
</tr>
<tr>
<td>15.1</td>
<td>Semantics</td>
<td>55</td>
</tr>
<tr>
<td>15.1.1</td>
<td>Execution Histories</td>
<td>55</td>
</tr>
<tr>
<td>15.1.2</td>
<td>Synchronization Object Types</td>
<td>56</td>
</tr>
<tr>
<td>15.2</td>
<td>Soundness</td>
<td>59</td>
</tr>
<tr>
<td>15.3</td>
<td>Derived Rules</td>
<td>73</td>
</tr>
<tr>
<td>15.4</td>
<td>Client Assertions</td>
<td>75</td>
</tr>
</tbody>
</table>

All the cross references are hyper-linked.
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 version of the TL2 commit procedure. This specification has immediately derived from the specification.

Figure 12. Example Specification

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 version of the TL2 commit procedure. This specification has derived from the 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 \( \triangleright x \leftarrow o.n.(u) \) 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 maps l to the receiving object o, name maps l to the method name n, thread maps l to the calling thread identifier \( \tau \), arg1 maps l to the first argument u (that is either a variable x or a value v), and retv maps l to the return value x. The function cond 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(l) = true for every label l. Every specification \( \pi \) defines a program order \( \rightarrow \) on the labels. Intuitively, \( l_1 \rightarrow l_2 \) 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 \( \rightarrow \) simply represents the order of labels in the program. We postpone relaxed order of method calls to next later section.
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.** \( \pi \cdot (\text{arg1}(R_1) < \text{arg1}(R_2)) \Rightarrow (R_1 < R_2) \).

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 \( \pi \) 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 \( \pi \), 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 \( \pi \), we have that (5) \( C_1 \) and \( C_2 \) are iaf method calls on lock 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 \( \pi \), 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 \( \pi \), 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 [15], we have that (14) an unlock method call by \( T_1 \) is linearized before a lock method call by \( T_2 \). From \( \pi \), 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 \( \pi \), 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 \( \pi \) is a specification, \( \Gamma \) 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 \( \Gamma \), the specification \( \pi \) has the property \( \mathcal{A} \). The assertions are first-order logic assertions that involve the unary predicate \( \text{exec} \), the binary predicates \( \prec \) (execution order) and \( \prec_o \) (linearization order of linearizable object \( o \)) and functions \( \text{obj}, \text{name}, \text{thread}, \text{arg1} \) and \( \text{retv} \). The assertion \( \text{exec}(l) \) 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
The execution order of two method calls 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 \( \pi \) 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 \( \pi \), yet the soundness of the logic makes us able to prove it by reasoning about \( \pi \) 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 \( \text{Control} \) states that a method call is executed if and only if its enclosing condition is satisfied. The introduction rule \( \text{In} \) 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 \( \text{P2X} \) 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 \( \text{Snc} \) intuitively states that every executed method originates from a call site in the specification. Let \( \text{Calls}_o(a,n) \) denote the set of labels of call sites where method name \( n \) is called on the object name \( o \) in the specification \( \pi \). 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 \( \text{Calls}_o(a,n) \). For presentation purposes, this small example does not involve procedure calls and hence the rules \( \text{Control}, \text{In}, \) and \( \text{Snc} \) are simplified.

The rule \( \text{X2L} \) states the real-time-preservation property. The execution order of two method calls on a linearizable object is preserved in the linearization order. \( \text{LT} \) denotes the set of linearizable object types. The rule \( \text{XLTrans} \) states the execution-linearization-transitivity property defined above. Similarly, the rule \( \text{LockUnlockPair} \) and the rule \( \text{CountSeq} \) state the lock-unlock-pair and count-sequence properties defined above. The rule \( \text{LockUnlockPair} \) 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 \( \text{CountSeq} \) 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

\[
\pi, \Gamma \vdash arg1(R_1) < arg1(R_2)
\]

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

\[
\pi, \Gamma \vdash R_1 < R_2
\]

From 18, we have

\[
\pi, \Gamma \vdash arg1(R_1) < arg1(R_2)
\]

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 \( \text{Control} \), we have

\[
\pi, \Gamma \vdash \text{exec}(L_1)
\]

\[
\pi, \Gamma \vdash \text{exec}(C_1)
\]

\[
\pi, \Gamma \vdash \text{exec}(U_1)
\]

\[
\pi, \Gamma \vdash \text{exec}(C_2)
\]

\[
\pi, \Gamma \vdash \text{exec}(U_2)
\]

From the rule \( \text{In} \) on 23, 27, 22, 26, and the specification \( \pi \), we have

\[
\pi, \Gamma \vdash \text{arg1}(R_1) = v_1
\]

\[
\pi, \Gamma \vdash \text{arg1}(R_2) = v_2
\]

\[
\pi, \Gamma \vdash \text{retv}(C_1) = v_1
\]

\[
\pi, \Gamma \vdash \text{retv}(C_2) = v_2
\]

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

\[
\pi, \Gamma \vdash \text{arg1}(R_1) = \text{retv}(C_1)
\]

\[
\pi, \Gamma \vdash \text{arg1}(R_2) = \text{retv}(C_2)
\]

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

\[
\pi, \Gamma \vdash \text{retv}(C_1) < \text{retv}(C_2)
\]

From the rule \( \text{In} \) on 22, and the specification \( \pi \), we have

\[
\pi, \Gamma \vdash \text{obj}(C_1) = \text{clock}
\]

\[
\pi, \Gamma \vdash \text{name}(C_1) = \text{iaf}
\]

By the rule \( \text{In} \) on 26, and the specification \( \pi \), we have

\[
\pi, \Gamma \vdash \text{obj}(C_2) = \text{clock}
\]

From rule \( \text{CountSeq} \) on 22, 36, 37, 26, 38, 35, we have

\[
\pi, \Gamma \vdash C_1 \prec_{clock} C_2
\]
that is $C_1$ is linearized before $C_2$. The next step is to use rule P2X. From $\pi$, we have

$$L_1 \rightarrow_{\pi} C_1 \quad (40)$$

$$C_2 \rightarrow_{\pi} U_2 \quad (41)$$

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

$$\pi, \Gamma \vdash L_1 < C_1 \quad (42)$$

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

$$\pi, \Gamma \vdash C_2 < U_2 \quad (43)$$

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

$$\pi, \Gamma \vdash L_1 < U_2 \quad (44)$$

By the rule In on 21, and the specification $\pi$, we have

$$\pi, \Gamma \vdash \text{obj}(L_1) = \text{lock} \quad (45)$$

$$\pi, \Gamma \vdash \text{name}(L_1) = \text{lock} \quad (46)$$

$$\pi, \Gamma \vdash \text{thread}(L_1) = T_1 \quad (47)$$

Similarly, by the rule In on 28, and the specification $\pi$, we have

$$\pi, \Gamma \vdash \text{obj}(U_2) = \text{lock} \quad (48)$$

$$\pi, \Gamma \vdash \text{name}(U_2) = \text{unlock} \quad (49)$$

$$\pi, \Gamma \vdash \text{thread}(U_2) = T_2 \quad (50)$$

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

$$\pi, \Gamma \vdash L_1 <_{\text{lock}} U_2 \quad (51)$$

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.

$$\pi, \Gamma \vdash \text{isOwnerRespecting} \pi) \quad (52)$$

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

$$\pi, \Gamma \vdash \text{isLock}_{\text{lock}}(L_1) \quad (53)$$

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

$$\pi, \Gamma \vdash \text{isUnlock}_{\text{lock}}(U_2) \quad (54)$$

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

$$\pi, \Gamma \vdash \exists \ell_{u_1}, \ell_{l_2} : \text{isUnlock}_{\text{lock}}(\ell_{u_1}) \land \text{thread}(\ell_{u_1}) = T_1 \land \text{isLock}_{\text{lock}}(\ell_{l_2}) \land \text{thread}(\ell_{l_2}) = T_2 \land \ell_{u_1} <_{\text{lock}} \ell_{l_2} \quad (55)$$

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

$$\pi, \Gamma \vdash \text{isUnlock}_{\text{lock}}(l_{u_1}) \quad (56)$$

$$\pi, \Gamma \vdash \text{thread}(l_{u_1}) = T_1 \quad (57)$$

$$\pi, \Gamma \vdash \text{isLock}_{\text{lock}}(l_{l_2}) \quad (58)$$

$$\pi, \Gamma \vdash \text{thread}(l_{l_2}) = T_2 \quad (59)$$

$$\pi, \Gamma \vdash l_{u_1} <_{\text{lock}} l_{l_2} \quad (60)$$

From the definition of isUnlock on 56, we have

$$\pi, \Gamma \vdash \text{exec}(l_{u_1}) \quad (61)$$

$$\pi, \Gamma \vdash \text{obj}(l_{u_1}) = \text{lock} \quad (62)$$

$$\pi, \Gamma \vdash \text{name}(l_{u_1}) = \text{unlock} \quad (63)$$

From $\pi$, we have

$$\text{Calls}_{\pi} \{\text{lock, unlock}\} = \{U_1, U_2\} \quad (64)$$

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

$$\pi, \Gamma \vdash l_{u_1} = T_1 \lor l_{u_1} = T_2 \quad (65)$$

Using negation introduction, from 50 and 57, we have

$$\pi, \Gamma \vdash \neg l_{u_1} = T_2 \quad (66)$$

By disjunction syllogism on 65 and 66, we have

$$\pi, \Gamma \vdash l_{u_1} = T_1 \quad (67)$$

Similarly, using the rule Src, we can show that

$$\pi, \Gamma \vdash l_{l_2} = T_2 \quad (68)$$

By substitution of 67 and 68 to 60, we have

$$\pi, \Gamma \vdash U_1 <_{\text{lock}} L_2 \quad (69)$$

From $\pi$, we have

$$R_1 \rightarrow_{\pi} U_1 \quad (70)$$

$$L_2 \rightarrow_{\pi} R_2 \quad (71)$$

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

$$\pi, \Gamma \vdash R_1 < U_1 \quad (72)$$

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

$$\pi, \Gamma \vdash L_2 < R_2 \quad (73)$$

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

$$\pi, \Gamma \vdash R_1 < R_2 \quad (74)$$
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`.

\[
\begin{align*}
& c \triangleright \text{foreach } (i \in \text{set}) \\
& \text{s}
\end{align*}
\]

Let `b` be a fresh variable name. We define `slIter(s,i)`, the `i`th iteration, as follows:

\[
\begin{align*}
\text{slIter}(s,i) &= c; b = \text{set.contains}(i), \\
& \text{if } (b_i) \\
& \text{sIndexed}(s,i)
\end{align*}
\]

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

\[
\begin{align*}
\text{slIter}(s,0), \\
\text{slIter}(s,1), \\
\text{slIter}(s,2), \\
\vdots \\
\text{slIter}(s,\text{max})
\end{align*}
\]

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`.

\[
\begin{align*}
& c \triangleright \text{foreach } ((i,v) \in \text{map}) \\
& \text{s}
\end{align*}
\]

We define `mlIter(s,i)`, the `i`th iteration, as follows:

\[
\begin{align*}
\text{mlIter}(s,i) &= c; v_i = \text{map.get}(i), \\
& \text{if } (v_i \neq 0) \\
& \text{mIndexed}(s,i)
\end{align*}
\]

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

\[
\begin{align*}
\text{mlIter}(s,0), \\
\text{mlIter}(s,1), \\
\text{mlIter}(s,2), \\
\vdots \\
\text{mlIter}(s,\text{max})
\end{align*}
\]

where `max` is the maximum key.

Transaction Syntax. A transactional memory description \(\pi_{TM}\) is a particular case of an algorithm description \((T, D_{TM}, P_{TM})\) where

\[
\begin{align*}
D_{TM} &= \text{def } \text{init}(s_0, r_0), \\
& \text{def } \text{read}(i, s_1, r_1), \\
& \text{def } \text{write}(i, v, s_2, r_2), \\
& \text{def } \text{commit}(s_3, r_3), \\
& \text{d}^* \\
P_{TM} &= \text{trans}_0(\text{trans}_1 || \text{trans}_2 || ... || \text{trans}_n)
\end{align*}
\]

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(i)`, `read(i)`, `write(i, v)` and `commit(i)`. The three specific values `C`, `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(i)` initializes the transaction `t`. The method `read(i)` returns the value of location `i` or `A` (if the transaction is aborted). The method `write(i, v)` writes `v` to location `i` and returns `ok` (if the write is successful) or returns `A` (if the transaction is aborted). The method `commit(i)` tries to commit transaction `t` and returns `C` (if the transaction is successfully committed) or returns `A` (if it is aborted). \(P_{TM}\) is an arbitrary client transaction. The initializing transaction \(\text{trans}_0\) initializes every location to zero. It is the sequence of `init(i)`, `write(i, 0)` method calls for every location `i` and then `commit(i)`. Each transaction \(\text{trans}_j\) `1 \leq j \leq n` starts with `init(i)` and then invokes a sequence of `read(i)` and `write(i, v)` method calls (for arbitrary location `i` and arbitrary value `v`). It stops invoking method calls if it receives abortion `A` from the previous method call. It finally invokes `commit(i)` if it is not already aborted. Let \(\Pi_{TM}\) 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 `j01` 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 \(\text{trans}_0\) that initializes every location to zero is defined as follows:

\[
\text{trans}_0 := IL_0 \triangleright \text{init}_0(); \\
\text{c}_0 \triangleright \text{write}_0(0, 0); \\
\text{c}_1 \triangleright \text{write}_0(1, 0); \\
\vdots \\
\text{c}_m \triangleright \text{write}_0(m, 0); \\
CL_0 \triangleright \text{commit}_0()
\]
Each transaction \( trans_j \) \( 1 \leq j \leq n \) is defined as follows:

\[
trans_j := \text{IL}_j \triangleright init_j(); \quad (82)
\]

\[
op_j
\]

\[
op_j := \begin{cases} 
\text{c} \triangleright x = \text{read}_j(v_1, v_2); \\
\quad \text{if } \neg(x = A) \\
\text{c} \triangleright x = \text{write}_j(v); \\
\quad \text{if } \neg(x = A) \\
\text{CL}_j \triangleright \text{commit}_j();
\end{cases}
\]

**Well-formedness.** The \( \text{init} \) method returns \( \text{ok} \). The \( \text{read} \) method does not return \( \text{ok} \) or \( C \). The \( \text{write} \) method does not return \( C \). The \( \text{commit} \) method either returns \( C \) or \( A \).

\[ \forall c \in \text{Returns}_\text{init}(\text{init}): \arg_1(\pi)(c) = \text{ok} \]

\[ \forall c \in \text{Returns}_\text{read}(\text{read}): \arg_1(\pi)(c) \neq \text{ok} \land \arg_1(\pi)(c) \neq C \]

\[ \forall c \in \text{Returns}_\text{write}(\text{write}): \arg_1(\pi)(c) \neq C \]

\[ \forall c \in \text{Returns}_\text{commit}(\text{commit}): \arg_1(\pi)(c) = C \lor \arg_1(\pi)(c) = A. \]

In addition, it is assumed that in every execution of the transaction \( trans_0 \), all the \( \text{write} \) method calls return \( \text{ok} \).

Let \( \Pi_{TM} \) denote the set of transactional memory specifications.

We define two functions \( \text{initOf} \) and \( \text{commitOf} \) that map a thread value to its initialization and commitment labels.

\[
\text{initOf}(T) = \text{IL}_T \quad (83)
\]

\[
\text{commitOf}(T) = \text{CL}_T \quad (84)
\]
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 \( l <_X l' \) then \( \neg(l' <_X l) \land \neg(l' \sim_X l) \land \neg(l' = l) \)

Lemma 10.2 (XTrans). For every execution history \( X \) and method calls \( l \), \( l' \), and \( l'' \), if \( l <_X l' \) and \( l' <_X l'' \) then \( l <_X l'' \)

Lemma 10.3 (XXTrans). For every execution history \( X \) and method calls \( l_1 \), \( l_2 \), \( l_3 \), and \( l_4 \), if \( l_1 <_X l_2 \), \( l_2 \leq_X l_3 \), and \( l_3 <_X l_4 \) then \( l_1 <_X l_4 \)

Lemma 10.4 (XTOTAL). For every execution history \( X \) and method calls \( l \) and \( l' \), if \( l \in X \) and \( l' \in X \), then \( (l <_X l') \lor (l' <_X l) \lor (l \sim_X l') \lor (l = l') \)

Lemma 10.5 (X2X). For every execution history \( X \) and method calls \( l \) and \( l' \), if \( l <_X l' \) 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 \( l <_X l' \) and \( \text{inv}(l') <_X \text{inv}(l'') \) then \( l <_X l'' \).

Lemma 10.7 (RX2X). For every execution history \( X \) and method calls \( l \), \( l' \), and \( l'' \) if \( \text{ret}(l) <_X \text{ret}(l') \) and \( l <_X l'' \) then \( l <_X l'' \).
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, HB(o), is a set of histories that is constrained as follows:

\[ HB(o) \cap \text{Sequential} \subseteq \text{SeqSpec}(o) \] (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 \(<_L\) is a sequentialization order of X. The semantics of a sequentially-consistent object o, HL(o), is defined as the following set of execution and sequentialization pairs.

\[ HL(o) = \{ (X, L) \mid X \equiv L \land L \in \text{SeqSpec}(o) \land \forall T \in X : \langle X, T \rangle \subseteq \langle L \rangle \} \] (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 \(<_L\) is a linearization order of X. The semantics of a linearizable object o, HL(o), is defined as the following set of execution and linearization pairs.

\[ HL(o) = \{ (X, L) \mid X \equiv L \land L \in \text{SeqSpec}(o) \land \langle X \rangle \subseteq \langle L \rangle \} \] (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 \(<_X\) l’ then l \(<_L\) 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 \(<_L\) l’ then l \(\leq_X\) l’.

Lemma 10.13 (LASYM). For every serialization or linearization L of an execution history X on object o and method calls l and l’, if l \(<_L\) l’ then \(\sim_l(l’ <_L l) \land \sim_l(l = l’)

Lemma 10.14 (LTRANS). For every serialization or linearization L of an execution history X on object o and method calls l, l’, and l”, if l \(<_L\) l’ and l’ \(<_L\) l” then l \(<_L\) l”.

Lemma 10.15 (LTOTAL). For every serialization 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 \(<_L\) l’) \lor (l’ \(<_L\) l) \lor (l = l’).

Lemma 10.16 (L2X). For every serialization or linearization L of an execution history X on object o and method calls l and l’, if (l \(<_L\) 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. \(\forall l: p\) can be rewritten as \(\bigwedge \{l \in \text{Labels}(X)\} p(X)\) and \(\exists l: p\) can be rewritten as \(\bigvee \{l \in \text{Labels}(X)\} p(X)\).
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 \( l_1 \prec_X l_2, l_2 \prec_X l_3, l_3 \prec_X l_4 \), then \( l_1 \prec_X l_4 \).

See section 15.1.2 for proofs.

10.2.1 Register

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

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

\[
\text{isXRead}_{X,r}(l_r) = l_r \in X \land \text{obj}_X(l_r) = r \land \text{name}_X(l_r) = \text{read}
\]

\[
\text{isXWrite}_{X,r}(l_w) = l_w \in X \land \text{obj}_X(l_w) = r \land \text{name}_X(l_w) = \text{write}
\]

\[
\text{NoWriteBetween}_{X,r}(l_w, l_r) = \forall l'_w : \text{isXWrite}_{X,r}(l'_w) \Rightarrow (l'_w \preceq_X l_w \lor l_r \prec_X l'_w)
\]

\[
\text{isXWrite}_{X,r}(l_w, l_r) = \text{isXWrite}_{X,r}(l_w) \land l_w \prec_X l_r \land \text{NoWriteBetween}_{X,r}(l_w, l_r)
\]

\[
\text{Legal}(r) = \{ S | \forall l_r : \text{isXRead}_{X,r}(l_r) \Rightarrow \exists l_w : \text{isXWrite}_{X,r}(l_w, l_r) \land \text{retu}_{\text{r}}(l_r) = \text{arg}1_{\text{s}}(l_w) \}
\]

\[
\text{SeqSpec}(r) = \{ S | S[r = S \land S \in \text{Sequential} \cap \text{Legal}(r)] \}
\]

Basic Register. A basic register is a basic instance of the register type. Let \( \text{BasicRegister} \) denote the type of basic registers.

Lemma 10.19. In every sequential execution on a basic register, every \( \text{read} \) reads the value that the latest preceding \( \text{write} \) writes. Formally,

\[
\forall \text{reg} \in \text{BasicRegister} : \forall X \in \mathbb{H}_B(\text{reg}) : X \in \text{Sequential} \Rightarrow \forall l_r : \text{isXRead}_{X,\text{reg}}(l_r) \Rightarrow \\
\exists l_w : \text{isXWrite}_{X,\text{reg}}(l_w, l_r) \land \text{retu}_{\text{X}}(l_r) = \text{arg}1_{\text{s}}(l_w)
\]

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

\[
\text{isXRaceFree}_{X,r}(l) = \forall l_w : \text{isXWrite}_{X,r}(l_w) \Rightarrow l_w \preceq_X l \lor l \prec_X l_w
\]

\[
\text{isXSequentiallyWritten}_{X,r}(X) = \forall l \in X : \text{isXWrite}_{X,r}(l) \Rightarrow \text{isXRaceFree}_{X,r}(l)
\]

A method call is race-free if an only if there is no \( \text{write} \) method call that executes concurrent to it. An execution is sequentially-written if and only if every pair of \( \text{write} \) method calls on it are ordered in the execution order or in other words, every \( \text{write} \) 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 \( \text{race-free} \) \( \text{read} \) reads the value that the latest preceding \( \text{write} \) writes. The semantics of a basic register \( r \), \( \mathbb{H}_B(r) \), is defined as follows:

\[
\mathbb{H}_B(r) = \{ X | X[o = X] \land \text{isXSequentiallyWritten}_{X,r}(X) \Rightarrow \\
\forall l_r : \text{isXRead}_{X,r}(l_r) \land \text{isXRaceFree}_{X,r}(l_r) \Rightarrow \\
\exists l_w : \text{isXWrite}_{X,r}(l_w, l_r) \land \text{retu}_{\text{x}}(l_r) = \text{arg}1_{\text{s}}(l_w) \}
\]
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,

\[
\forall \text{reg} \in \text{BasicRegister}: \forall X \in \mathbb{H}_B(\text{reg}) : \text{isXSequentiallyWritten}_r(X) \Rightarrow
\]

\[
\forall l_r: \text{isXRead}_X,_r(l_r) \land \text{isXRaceFree}_X,_r(l_r) \Rightarrow
\]

\[
\exists l_w: \text{isXWriter}_X,_r(l_w, l_r) \land
\]

\[
\text{ret}_X(l_r) = \text{arg}1_X(l_w)
\]

**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,

\[
\forall r \in \text{AtomicRegister}: \forall (X, l) \in \mathbb{H}_L(r):
\]

\[
\forall l_r: \text{isXRead}_X,_r(l_r) \Rightarrow
\]

\[
\exists l_w: \text{isXWriter}_X,L_r(l_w, l_r) \land
\]

\[
\text{ret}_X(l_r) = \text{arg}1_X(l_w)
\]

**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.

Let us define

\[
\text{LNoWriteBetween}_X,L_r(l_w, l_r) = \forall l_w': \text{isXWrite}_X,l_r(l_w') \Rightarrow (l_w' \preceq l_w \lor l_r \prec l_w')
\]

(99)

\[
\text{isLWriter}_X,L_r(l_w, l_r) = \text{isXWrite}_X,l_r(l_w) \land
\]

\[
l_w \preceq l_r \land
\]

\[
\text{LNoWriteBetween}_X,L_r(l_w, l_r)
\]

Let us define

Consider the following four concurrent threads.

\[
T_1 \rightarrow \text{r1.write(1)} \parallel T_2 \rightarrow \text{r2.write(1)} \parallel T_3 \rightarrow \text{x1 = r1.read()} \parallel T_4 \rightarrow \text{y2 = r2.read()}
\]

\[
L_{31} \rightarrow \text{L32} \parallel L_{41} \rightarrow \text{L42}
\]

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) \( L_{r_1} \rightarrow \text{L42} \rightarrow x_2 = \text{r2.read()} \rightarrow \text{r1.write(1)} \rightarrow L_{31} \rightarrow x_1 = \text{r1.read()} \)

for \( r_1 \) and the sequentialization order

(2) \( L_{r_2} \rightarrow \text{L32} \rightarrow y_2 = \text{r2.read()} \rightarrow \text{L21} \rightarrow y_1 = \text{r2.read()} \rightarrow L_{41} \rightarrow y_2 = \text{r2.read()} \)

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) \( L_{31} \preceq X L_{32} \), (4) \( L_{41} \preceq X L_{42} \). By X2L’ on 2, we have (5) \( L_{32} \preceq X L_{41} \). By X2TRANS on 3, 5 and 4, we have (6) \( L_{31} \preceq X L_{42} \). By X2L on 6, we have \( L_{31} \preceq X_{r_1} L_{42} \) 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 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 v₁ returns true if the written value of the latest preceding successful write is v₁ 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

\[
\begin{align*}
isXCAS_{X,r}(l_W) & = l_W \in X \land obj_X(l_W) = r \land name_X(l_W) = cas \quad (102) \\
isXCasWrite_{X,r}(l_W) & = isXWrite(l_W) \lor (isXCAS(l_W) \land retv_X(l_W) = true) \quad (103) \\
writtenValue_X(l_W) & = \begin{cases} 
arg1_X(l_W) & \text{if } name_X(l_W) = write \\
arg2_X(l_W) & \text{if } name_X(l_W) = cas 
\end{cases} \quad (104) \\
L_{NoWriteBetween_{X,L,r}}(l_W,l_R) & = \forall l_W': isXCasWrite_{X,r}(l_W') \Rightarrow (l_W' \preceq l_R \lor l_W' \preceq l_R) \quad (105) \\
isLCasWrite_{X,L,r}(l_W,l_R) & = isXCasWrite_{X,r}(l_W) \land l_w < l_R \land L_{NoWriteBetween_{X,L,r}}(l_W,l_R) \\
\end{align*}
\]

**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,

\[
\begin{align*}
\forall r \in \text{AtomicCASRegister} : \forall (X,L) \in \mathbb{H}_L(r) : \\
\forall l_R : isXRead_{X,r}(l_R) \Rightarrow \\
\exists l_W : isLCasWrite_{X,L,r}(l_W,l_R) \land \\
retv_X(l_R) = arg1_X(l_W) 
\end{align*}
\]

**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,

\[
\begin{align*}
\forall reg \in \text{AtomicCASRegister} : \forall (X,Reg) \in \mathbb{H}_L(reg) : \\
\forall l_C,l_W : \\
isXCAS_{X,reg}(l_C) \land \\
isLCasWrite_{X,reg}(l_W,l_R) \\
\Rightarrow \\
\left( \text{writtenValue}_X(l_W) = arg1_X(l_C) \Rightarrow retv_X(l_C) = true \right) \land \\
\left( \neg (\text{writtenValue}_X(l_W) = arg1_X(l_C)) \Rightarrow retv_X(l_C) = false \right) 
\end{align*}
\]

10.2.3 Lock

Abstract lock. An abstract lock l is an object that encapsulates a state, acquired A or released R, and supports the following methods: **lock**: The method call l.lock() changes the state from R to A. **unlock**: The method call l.unlock() changes the state from A to R. **read**: The method call l.read() returns true if the state of lock is 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.

\[
isXLock_{X,lo}(l) = \begin{cases} \text{true} & \text{if } l \in X \land \text{obj}_X(l) = \text{lo} \land \text{name}_X(l) = \text{lock} \\ \text{false} & \text{otherwise} \end{cases}
\]

(109)

\[
isXUnlock_{X,lo}(l) = \begin{cases} \text{true} & \text{if } l \in X \land \text{obj}_X(l) = \text{lo} \land \text{name}_X(l) = \text{unlock} \\ \text{false} & \text{otherwise} \end{cases}
\]

(110)

\[
isXRead_{X,lo}(l) = \begin{cases} \text{true} & \text{if } l \in X \land \text{obj}_X(l) = \text{lo} \land \text{name}_X(l) = \text{read} \\ \text{false} & \text{otherwise} \end{cases}
\]

(111)

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.

\[
isXOwnerRespecting_{lo}(X) = \forall l: isXUnlock_{X,lo}(l) \Rightarrow \exists l': isXLock_{X,lo}(l') \land thread_X(l') = thread_X(l) \land l' \prec_X l \land \forall l'': (isXUnlock_{X,lo}(l'') \land thread_X(l'') = thread_X(l)) \Rightarrow (l'' \prec_X l' \lor l \prec_X l'')
\]

(112)

**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,

\[
\forall o \in \text{Lock} : \forall (X, L) \in \mathbb{H}_L(o) : \forall l_1, l_2 : (isXOwnerRespecting_{o}(X) \land isXLock_{X,o}(l_1) \land isXUnlock_{X,o}(l_2) \land l_1 \prec_L l_2) \Rightarrow \exists l_{u1}, l_{u2} : (isXUnlock_{X,o}(l_{u1}) \land thread_X(l_{u1}) = thread_X(l_1) \land isXLock_{X,o}(l_{u2}) \land thread_X(l_{u2}) = thread_X(l_2) \land l_{u1} \prec_L l_{u2})
\]

(113)

**Lemma 10.30 (LockReadL).** In an owner-respecting execution for a lock \( l \), if a read method call that returns \( \text{false} \) 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,

\[
\forall o \in \text{Lock} : \forall (X, L) \in \mathbb{H}_L(o) : \forall l_{u1}, l_{r2} : (isXOwnerRespecting_{o}(X) \land isXRead_{X,o}(l_{r2}) \land \text{ret}_X(l_{r2}) = \text{false} \land isXUnlock_{X,o}(l_{u1}) \land l_{r2} \prec_L l_{u1}) \Rightarrow \exists l_{l1} : (isXLock_{X,o}(l_{l1}) \land thread_X(l_{l1}) = thread_X(l_{u1}) \land l_{r2} \prec_L l_{l1})
\]

(114)
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,

\[ \forall o \in \text{Lock}: \forall (X, L) \in \mathbb{H}_L(o): \forall l_1, l_2: \]
\[ (\text{isXOwnerRespecting}_o(X) \land \text{isXLock}_{X, o}(l_1) \land \text{isXRead}_{X, o}(l_2) \land \text{retv}_X(l_2) = \text{false} \]
\[ l_1 \preceq_L l_2 \implies \exists l_{u_1}: \]
\[ \text{isXUnlock}_{X, o}(l_{u_1}) \land \text{thread}_X(l_{u_1}) = \text{thread}_X(l_1) \land \]
\[ l_{u_1} \preceq_L l_2 \]

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,

\[ \forall o \in \text{Lock}: \forall (X, L) \in \mathbb{H}_L(o): \forall l_1, l_{u_1}, l_2: \]
\[ (\text{isXOwnerRespecting}_o(X) \land \text{isXLock}_{X, o}(l_1) \land \text{isXUnlock}_{X, o}(l_{u_1}) \land \text{thread}_X(l_{u_1}) = \text{thread}_X(l_1) \land \]
\[ \forall l_{u_1}': (\text{isXUnlock}_{X, o}(l_{u_1}') \land \text{thread}_X(l_{u_1}') = \text{thread}_X(l_{u_1})) \implies (l_{u_1}' \prec_X l_1 \lor l_{u_1} \preceq_X l_{u_1}') \land \text{isXRead}_{X, o}(l_{u_1}') \land \]
\[ l_1 \preceq_L l_2 \land l_{u_1}' \preceq_L l_{u_1} \]
\[ \implies \text{retv}_X(l_2) = \text{true} \]

10.2.4 Try-lock

Abstract Try-lock. A try-lock l is an object that encapsulates an abstract state, acquired A or released R, and in addition to lock, unlock and read methods, it supports the trylock method. If the state of the lock is R, l.trylock() changes it to 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 methods 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.

\[ \text{isXTryLock}_{X, o}(l) = \]
\[ l \in X \land \text{obj}_X(l) = o \land \text{name}_X(l) = \text{tryLock} \]
\[ \text{isXTLock}_{X, o}(l) = \]
\[ \text{isXLock}_{X, o}(l) \lor (\text{isXTryLock}_{X, o}(l) \land \text{retv}_X(l) = \text{true}) \]

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.

\[
\text{isXTOwnerRespecting}_o(X) = \forall l: \text{isXUnlock}_{X,o}(l) \Rightarrow \exists l': \text{isXTLock}_{X,o}(l') \land \\
l' <_X l \land \\
\forall l'': (\text{isXUnlock}_{X,o}(l'') \land \text{thread}_X(l'') = \text{thread}_X(l)) \Rightarrow l'' \prec_X l' \lor l \leq_X l'
\]

**Lemma 10.34.** If \(l\) is a try-lock, \(X\) is an owner-respecting history of \(I\) 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,

\[
\forall o \in \text{TryLock} : \forall (X,L) \in \mathbb{H}_L(o) : \forall l_{i_1}, l_{i_2} : \forall o_{l_{i_1}, l_{i_2}} :
\]

\[
(\text{isXTOwnerRespecting}_o(X) \land \\
\text{isXTLock}_{X,o}(l_{i_1}) \land \\
\text{isXUnlock}_{X,o}(l_{i_2}) \land \\
l_{i_1} \prec_L l_{i_2} \Rightarrow \\
\exists o_{l_{i_1}, l_{i_2}}:
\]

\[
\text{isXUnlock}_{X,o}(l_{i_1}) \land \text{thread}_X(l_{i_1}) = \text{thread}_X(l_{i_2}) \land \\
\text{isXTLock}_{X,o}(l_{i_2}) \land \text{thread}_X(l_{i_2}) = \text{thread}_X(l_{i_1}) \land \\
l_{i_1} \prec_L l_{i_2}
\]

**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,

\[
\forall o \in \text{TryLock} : \forall (X,L) \in \mathbb{H}_L(o) : \forall l_{u_1}, l_{r_2}:
\]

\[
(\text{isXTOwnerRespecting}_o(X) \land \\
\text{isXRead}_{X,o}(l_{r_2}) \land \text{retv}_X(l_{r_2}) = \text{false} \\
\text{isXUnlock}_{X,o}(l_{u_1}) \land \\
l_{r_2} \prec_L l_{u_1} \Rightarrow \\
\exists o_{l_{u_1}, l_{r_2}}:
\]

\[
\text{isXTLock}_{X,o}(l_{u_1}) \land \text{thread}_X(l_{u_1}) = \text{thread}_X(l_{r_2}) \land \\
l_{u_1} \prec_L l_{r_2}
\]

**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,

\[
\forall o \in \text{TryLock} : \forall (X,L) \in \mathbb{H}_L(o) : \forall l_{i_1}, l_{r_2}:
\]

\[
(\text{isXTOwnerRespecting}_o(X) \land \\
\text{isXTLock}_{X,o}(l_{i_1}) \land \\
\text{isXRead}_{X,o}(l_{r_2}) \land \text{retv}_X(l_{r_2}) = \text{false} \\
l_{i_1} \prec_L l_{r_2} \Rightarrow \\
\exists o_{l_{i_1}, l_{r_2}}:
\]

\[
\text{isXUnlock}_{X,o}(l_{i_1}) \land \text{thread}_X(l_{i_1}) = \text{thread}_X(l_{r_2}) \land \\
l_{i_1} \prec_L l_{r_2}
\]
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,

\[
\forall o \in \text{TryLock} : \forall (X, L) \in \mathbb{H}_L(o) : \forall l_{t1}, l_{u1}, l_{t2} : \tag{123}
\]
\[
(isXOwnerRespecting_o(X) \land isXTLock_{X,o}(l_{t1}) \land isXUnlock_{X,o}(l_{u1}) \land thread_X(l_{t1}) = thread_X(l_{u1}) \land \forall l'_{u1} : (isXUnlock_{X,o}(l'_{u1}) \land thread_X(l_{t1}) = thread_X(l'_{u1})) \Rightarrow (l'_{u1} \prec_X l_{t1} \lor l_{u1} \preceq_X l'_{u1})
\]
\[
isXRead_{X,o}(l_{t2}) \land l_{t1} \neq l_{t2} \land l_{t2} \neq l_{u1}
\]
\[
\Rightarrow retv_X(l_{t2}) = true
\]

10.2.5 Sequence-lock

Abstract seq-lock. A seq-lock \( l \) is an object that encapsulates a number and an abstract state, acquired \( A \) or released \( R \). 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 \( A \) and \( false \) otherwise. The method call \( l.compareAndLock(n) \) compares the encapsulated number with \( n \) and if they are equal, changes the state from \( R \) to \( A \) 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 \( A \) to \( R \).

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,

\[ \forall c \in \text{SCounter}: \forall (X, C) \in \mathbb{H}_{L}(c): \forall l, l': \]

\[ l \in X \land l' \in X \land \text{name}_{X}(l') = \text{iaf} \land l <_{C} l' \]

\[ \Rightarrow \text{ret}_{X}(l) < \text{ret}_{X}(l') \]

10.2.7 Set

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

Definition 10.42. The sequential specification of a set \( s \) is the set of sequential histories of \text{add} and \text{contains} method calls on \( s \) where every \text{contains} method call returns \text{true} if there is a preceding \text{add} method call with the same argument, and returns \text{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 \( \text{BasicSet} \) denote the type of basic sets.

Let us define

\[
isXContains_{X,s}(l) = \]

\[ l \in X \land \text{obj}_{X}(l) = s \land \text{name}_{X}(l) = \text{contains} \]

\[
isXAdd_{X,s}(l) = \]

\[ l \in X \land \text{obj}_{X}(l) = s \land \text{name}_{X}(l) = \text{add} \]

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

\[
\forall s \in \text{BasicSet}: \forall X \in \mathbb{H}_{R}(s): X \in \text{Sequential} \Rightarrow \]

\[ \forall l_{c}: \text{isXContains}_{X,s}(l_{c}) \land \text{ret}_{X}(l_{c}) = \text{true} \Rightarrow \]

\[ \exists l_{a}: \text{isXAdd}_{X,s}(l_{a}) \land \]

\[ \text{arg}_{1}(l_{a}) = \text{arg}_{1}(l_{c}) \land l_{a} <_{X} l_{c} \]

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

\[
\forall s \in \text{BasicSet}: \forall X \in \mathbb{H}_{R}(s): X \in \text{Sequential} \Rightarrow \]

\[ \forall l_{c}, l_{a}: \]

\[ \text{isXContains}_{X,s}(l_{c}) \land \]

\[ \text{isXAdd}_{X,s}(l_{a}) \land \]

\[ \text{arg}_{1}(l_{a}) = \text{arg}_{1}(l_{c}) \land l_{a} <_{X} l_{c} \]

\[ \Rightarrow \text{ret}_{X}(l_{c}) = \text{true} \]

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: \text{put}: The method call \( m.\text{put}(k, v) \) adds or updates the mapping of the key \( k \) to the value \( v \) (\( v \neq \bot \)) in the map \( m \). \text{get}: The method call \( m.\text{get}(k) \) returns the value that the map \( m \) associates with the key \( k \). It returns \( \bot \) if \( m \) does not map \( k \).

Definition 10.45. The sequential specification of a map \( m \) is the set of sequential histories of \text{put} and \text{get} method calls on \( m \) where every \text{get} method call returns \( \bot \) if there is no preceding \text{put} method call with the same key argument; otherwise it returns the second argument of the latest preceding \text{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 \texttt{BasicMap} denote the type of basic maps.

Let us define

\[
isXGet_{X,m}(l) = \quad l \in X \land \text{obj}_X(l) = m \land \text{name}_X(l) = \text{get} \tag{129}
\]

\[
isXPut_{X,m}(l) = \quad l \in X \land \text{obj}_X(l) = m \land \text{name}_X(l) = \text{put} \tag{130}
\]

\[
isXPutter_{X,m}(l_p,l_d) \iff
\]

\[
isXPut_{X,m}(l_p) \land \text{arg}_1_X(l_p) = \text{arg}_1_X(l_g) \land l_p <_X l_g \land
\]

\[
\forall l'_p : isXPut_{X,m}(l'_p) \land \text{arg}_1_X(l'_p) = \text{arg}_1_X(l_g) \Rightarrow (l'_p <_X l_p \lor l_g <_X l'_p) \tag{133}
\]

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

\[
\forall m \in \texttt{BasicMap} : \forall X \in \mathbb{H}_B(m) : X \in \texttt{Sequential} \Rightarrow \tag{134}
\]

\[
\forall l_g : isXGet_{X,m}(l_g) \land \neg (\text{retv}_X(l_g) = \bot) \Rightarrow
\]

\[
\exists l_p : isPutter_{X,m}(l_p,l_g) \land
\]

\[
\text{arg}_2_X(l_p) = \text{retv}_X(l_g)
\]

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

\[
\forall m \in \texttt{BasicMap} : \forall X \in \mathbb{H}_B(m) : X \in \texttt{Sequential} \Rightarrow \tag{135}
\]

\[
\forall l_g,l_p : \quad isXGet_{X,m}(l_g) \land
\]

\[
isPutter_{X,m}(l_p,l_g) \land
\]

\[
\Rightarrow \quad \text{retv}_X(l_g) = \text{arg}_2_X(l_p)
\]
10.3 Transactional Histories

**Transactional Memory.** The transactional memory is a singleton object `mem` that encapsulates a set of locations where each location, `i ∈ I, I = {1, . . . , m}` encapsulates a value `v`. The object `mem` has five methods `initt()`, `readt(i)`, `writt(i, v)`, `comitt()` and `abortt()`. The parameter `t` is the invoking transaction identifier. The method call `initt()` initializes `t` and returns `ok`. The method call `readt(i)` returns the value of location `i` or aborts `t` and returns `A`. The method `writt(i, v)` writes `v` to location `i` and returns `ok` or aborts `t` and returns `A`. The method `comitt()` tries to commit transaction `t`. If `t` is successfully committed, it returns `C`; otherwise, it returns `A`. The method `abortt()` aborts `t` and returns `A`. The object `mem` can be implicit, that is `readt(i)` abbreviates `mem.readt(i)`. The reserved values `ok, A, 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_{\text{init}} · H’` with the following conditions. `H_{\text{init}}` is the following history that initializes every location to `v_0`. `H_{\text{init}} = l_0 ≫ \text{init}(t_0) · l_0 ≫ \text{write}(t_0, 1, v_0): ok · . . . · l_m ≫ \text{write}(t_m, m, v_0): ok · l_0 ≫ \text{commit}(t_0); C`. For every `T ∈ H’`, the history `H’\{T` is a prefix of `E.E’`. The event sequence `E` is the initialization method call `l ≫ \text{init}(t)` (for some `l`), and then a sequence of reads `l ≫ \text{read}(t, i): v` and writes `l ≫ \text{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 ≫ \text{read}(t, i)), ret(l ≫ A), (2) inv(l ≫ \text{write}(t, i, v)), ret(l ≫ A), (3) inv(l ≫ \text{commit}(t)), ret(l ≫ C), (4) inv(l ≫ \text{commit}(t)), ret(l ≫ A), or (5) inv(l ≫ \text{abort}(t)), ret(l ≫ A).` Let `\mathbb{T}_{\text{History}}` 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}(π)` denote the set of complete transaction histories that result from execution of transactions with `π`. 

NFM’19, May 2019, Mohsen Lesani
Inference Rules

In this section, we now present the inference rules. The judgements are of the form $\pi, \Gamma \vdash A$ read assertion $A$ is derived from the assumption assertions $\Gamma$ for the specification $\pi$. The context $\Gamma$ is defined as follows:

$$\Gamma ::= \cdot \mid \Gamma; A$$


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.

```
Figure 16. Classical Inference Rules
```
\begin{align*}
\text{DisjSyllL} & \quad & \text{DisjSyllR} & \quad & \text{CondElim}' \\
\pi, \Gamma \vdash A \lor A' & \quad & \pi, \Gamma \vdash A \lor A' & \quad & \pi, \Gamma \vdash A' \Rightarrow A' \\
\pi, \Gamma \vdash \neg A & \quad & \pi, \Gamma \vdash \neg A' & \quad & \pi, \Gamma \vdash \neg A' \\
\pi, \Gamma \vdash A' & \quad & \pi, \Gamma \vdash A & \quad & \pi, \Gamma \vdash \neg A
\end{align*}

\textbf{Figure 17. Derived Classical Inference Rules}

\begin{align*}
\text{LRefl} & \quad & \text{ERefl} & \quad & \text{Zero} \\
\pi, \Gamma \vdash l = l & \quad & \pi, \Gamma \vdash e = e & \quad & \pi, \Gamma \vdash \neg(1 = 0) \\
\pi, \Gamma \vdash l = l' & \quad & \pi, \Gamma \vdash e = e' & \quad & \pi, \Gamma \vdash A[l := l'] \\
\pi, \Gamma \vdash A & \quad & \pi, \Gamma \vdash A & \quad & \pi, \Gamma \vdash A[e := e']
\end{align*}

\textbf{Figure 18. Equivalence and Arithmetic Rules}

\begin{align*}
\text{LSym} & \quad & \text{LTrans} & \quad & \text{ESym} & \quad & \text{ETrans} \\
\pi, \Gamma \vdash l = l' & \quad & \pi, \Gamma \vdash l = l' & \quad & \pi, \Gamma \vdash e = e' & \quad & \pi, \Gamma \vdash e = e' \\
\pi, \Gamma \vdash l' = l & \quad & \pi, \Gamma \vdash l' = l'' & \quad & \pi, \Gamma \vdash e = e & \quad & \pi, \Gamma \vdash e' = e'' \\
\pi, \Gamma \vdash l' = l'' & \quad & \pi, \Gamma \vdash e = e & \quad & \pi, \Gamma \vdash e = e''
\end{align*}

\textbf{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 \text{In} 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 $\varsigma'c$ 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 $\varsigma$. Note that the pre-label $\varsigma$ is a constant $c'$ when the method call $c$ is executed inside the body of a \textit{this} method call annotated with $c'$. The pre-label $\varsigma$ is $\epsilon$ when $c$ is the annotation of a \textit{this} method call.

The rule \text{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 $\phi$ is executed, it is from one of the call sites where $n$ is called on $\phi$ in the program.

The rule \text{OControl} states when a \textit{this} method call is executed. A \textit{this} method call is executed if and only if its execution condition is satisfied.

The rule \text{IControl} states when a method call in the body of a \textit{this} method call is executed. A method call (annotated with) $c'$ in a \textit{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 \text{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 $\varsigma'c_1$ and $\varsigma'c_2$ are executed, then $\varsigma'c_1$ is executed before $\varsigma'c_2$.

The rule \text{OX2IX} states that the execution order of two \textit{this} method calls implies the execution order of method calls in their bodies. If a \textit{this} method call $c_1$ is executed before another \textit{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 \text{TSeq} states that every thread is sequential. Every two \textit{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 \text{Caller} states that if a \textit{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 \text{Callee} states that if a method call in the body of a \textit{this} method call is executed, then the \textit{this} method call is executed and the parameters and the arguments of the \textit{this} method call are equal.

The rule \text{Ret} states that if a return statement of the body of a \textit{this} method call is executed, then the \textit{this} method call is executed and the parameters and the arguments of the \textit{this} method call are equal and the return value of the \textit{this} method call is the value that the return statement returns.

The rule \text{TLocal} states that every two executed method calls on the same thread-local object are from the same thread.

The rule \text{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 \text{IX2OX} states that if two method calls in the body of two \textit{this} method calls execute in order by the same thread, then the two \textit{this} method calls execute in the same order.
\[\text{Id}\]
\[\begin{align*}
\text{obj}_\pi(c) &= \emptyset & \text{name}_\pi(c) &= n \\
\text{thread}_\pi(c) &= \tau & \text{arg}_\pi(c) &= u & \text{return}_\pi(c) &= x \\
\end{align*}\]
\[\pi, \Gamma \vdash \text{exec}(\zeta c)\]
\[\pi, \Gamma \vdash \text{obj}(\zeta c) = c' \emptyset \wedge name(\zeta c) = n \wedge \text{thread}(\zeta c) = \zeta' \tau \wedge arg(\zeta c) = c' u \wedge return(\zeta c) = \zeta' x\]

\[\text{Src}\]
\[\pi, \Gamma \vdash \text{exec}(\zeta c)\]
\[\pi, \Gamma \vdash \text{obj}(\zeta c) = \emptyset \quad \pi, \Gamma \vdash \text{name}(\zeta c) = n\]
\[\text{Calls}_\pi(\text{basename}(\theta), n) = \{\zeta c\}\]
\[\pi, \Gamma \vdash \bigwedge_{i=1..n} c = c_i\]

\[\text{OControl}\]
\[c \in \text{Labels}(\mathcal{P})\]
\[\pi, \Gamma \vdash \text{exec}(c) \iff \text{cond}_\pi(c)\]

\[\text{IControl}\]
\[\text{Labels}(\text{name}_\pi(c)) = \{\zeta c\}\]
\[\text{PreReturns}_\pi(\zeta c') = \{\zeta c\}\]
\[\forall c' \in \text{Labels}(\mathcal{P}) \quad \pi, \Gamma \vdash \text{exec}(c) \land \exists c_i \land c' \text{cond}_\pi(c_i) \land \nexists \text{exec}(c_i c)\]

\[\text{P2X}\]
\[c_1 \rightarrow_\pi c_2\]
\[\pi, \Gamma \vdash \text{exec}(\zeta c_1) \quad \pi, \Gamma \vdash \text{exec}(\zeta c_2)\]
\[\pi, \Gamma \vdash \zeta' c_1 < \zeta' c_2\]

\[\text{OX2IX}\]
\[\pi, \Gamma \vdash c_1 < c_2\]
\[\pi, \Gamma \vdash \text{exec}(c_1 c_3) \quad \pi, \Gamma \vdash \text{exec}(c_2 c_4)\]
\[\pi, \Gamma \vdash \zeta_1 c_3 < c_2' c_4\]

\[\text{TSReq}\]
\[\pi, \Gamma \vdash \text{exec}(l_1) \quad \pi, \Gamma \vdash \text{exec}(l_2)\]
\[\pi, \Gamma \vdash \text{thread}(l_1) = \text{thread}(l_2)\]
\[\pi, \Gamma \vdash \text{obj}(l_1) = \text{obj}(l_2) = \text{this} \lor \neg \text{obj}(l_1) = \text{this} \land \neg \text{obj}(l_2) = \text{this}\]
\[\pi, \Gamma \vdash l_1 < l_2 \lor l_2 < l_1 \lor l_1 = l_2\]

\[\text{TLocal}\]
\[\mathcal{J}(\text{basename}(o)) = \text{ThreadLocal st}\]
\[\pi, \Gamma \vdash \text{exec}(l_1) \land \text{exec}(l_2)\]
\[\pi, \Gamma \vdash \text{obj}(l_1) = \text{obj}(l_2) = o\]
\[\pi, \Gamma \vdash \text{thread}(l_1) = \text{thread}(l_2)\]

\[\text{TReal}\]
\[\pi, \Gamma \vdash \tau \leq \tau'\]
\[\pi, \Gamma \vdash \text{exec}(l) \land \text{thread}(l) = \tau\]
\[\pi, \Gamma \vdash \text{exec}(l') \land \text{thread}(l') = \tau'\]
\[\pi, \Gamma \vdash l < l'\]

\[\pi, \Gamma \vdash c_1 < c_2 \land c_1 = c_2\]

\[\pi, \Gamma \vdash \text{thread}(c_1 c_3) = \text{thread}(c_2 c_4)\]

\[\pi, \Gamma \vdash c_1 = c_2 \lor c_1 < c_2 \lor c_1 > c_2\]

\[\pi, \Gamma \vdash \text{exec}(c)\]

\[\pi, \Gamma \vdash \text{obj}(c) = \text{this} \land \text{name}(c) = n\]

\[\pi, \Gamma \vdash c' \text{ thread} \land c' x = \text{arg}(c) \land \bigwedge_{i=1..n} \text{exec}(c_i c_1) \land \text{arg}(c_1 c_1) = \text{return}(c)\]

\[\pi, \Gamma \vdash \text{obj}(c) = \text{this} \land \text{name}(c) = n\]

\[\pi, \Gamma \vdash c' \tau \land c' x = \text{arg}(c) \land \text{exec}(c) \land \text{return}(c) = \text{arg}(c)\]

\[\pi, \Gamma \vdash \text{obj}(c) = \text{this} \land \text{name}(c) = n\]

\[\pi, \Gamma \vdash c' \text{ thread} \land c' x = \text{arg}(c) \land \bigwedge_{i=1..n} \text{exec}(c_i c_1) \land \text{arg}(c_1 c_1) = \text{return}(c)\]

\[\pi, \Gamma \vdash \text{obj}(c) = \text{this} \land \text{name}(c) = n\]

\[\pi, \Gamma \vdash c' \tau \land c' x = \text{arg}(c) \land \text{exec}(c) \land \text{return}(c) = \text{arg}(c)\]

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

**Figure 21.** Derived Structure Inference Rules
The basic inference rules axiomatize the properties of the execution and linearization orders and their interdependence. The 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 \( l_4 \).

### 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 \textsc{XTotal} states the totality property of the precedence and concurrency execution relations. Every two method calls either execute in order or concurrently.

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

The rule \textsc{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 \textsc{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 \textsc{LTrans} states the transitivity property of linearization orders.

The rule \textsc{LTotal} states the totality property of linearization orders.

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

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

The rule \textsc{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 \textsc{X2L‘} states the contra-positive of the rule \textsc{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 $A_{\text{Reg}}$ 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 $isRaceFree(l)$ 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.

**Register Inference Rules.**

#### Figure 24.

<table>
<thead>
<tr>
<th>$A_{\text{Reg}}$</th>
<th>$B_{\text{Reg}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{T}_{\text{base}}(\text{reg}) = \text{AtomicRegister}$</td>
<td>$\mathcal{T}_{\text{base}}(\text{reg}) = \text{BasicRegister}$</td>
</tr>
<tr>
<td>$\pi, \Gamma \vdash isRead_{\text{reg}}(l_R)$</td>
<td>$\pi, \Gamma \vdash isSequentiallyWritten(reg)$</td>
</tr>
<tr>
<td>$\pi, \Gamma \vdash isWriter_{\text{reg}}(l_W, l_R)$</td>
<td>$\pi, \Gamma \vdash isRaceFree_{\text{reg}}(l_R)$</td>
</tr>
<tr>
<td>$isRead_{\tau}(l_R) \iff \pi, \Gamma \vdash isRead_{\text{reg}}(l_R)$</td>
<td>$\pi, \Gamma \vdash isRead_{\text{reg}}(l_R)$</td>
</tr>
<tr>
<td>$isWrite_{\tau}(l_W) \iff \pi, \Gamma \vdash isWrite_{\text{reg}}(l_W, l_R)$</td>
<td>$\pi, \Gamma \vdash isSequentiallyWritten(reg)$</td>
</tr>
<tr>
<td>$\pi, \Gamma \vdash \exists l_W : isWriter_{\text{reg}}(l_W, l_R) \land retv(l_R) = arg1(l_W)$</td>
<td>$\pi, \Gamma \vdash \exists l_W : isWriter_{\text{reg}}(l_W, l_R) \land retv(l_R) = arg1(l_W)$</td>
</tr>
</tbody>
</table>

#### Derived Register Inference Rules:

<table>
<thead>
<tr>
<th>$A_{\text{Reg}'}$</th>
<th>$B_{\text{Reg}'}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{T}_{\text{base}}(\text{reg}) = \text{BasicRegister}$</td>
<td>$\mathcal{T}_{\text{base}}(\text{reg}) = \text{BasicRegister}$</td>
</tr>
<tr>
<td>$\pi, \Gamma \vdash isRead_{\text{reg}}(l_R)$</td>
<td>$\pi, \Gamma \vdash isRead_{\text{reg}}(l_R)$</td>
</tr>
<tr>
<td>$\pi, \Gamma \vdash isSequentiallyWritten(reg)$</td>
<td>$\pi, \Gamma \vdash isSequentiallyWritten(reg)$</td>
</tr>
<tr>
<td>$\pi, \Gamma \vdash \exists l_W : isWriter_{\text{reg}}(l_W, l_R) \land retv(l_R) = arg1(l_W)$</td>
<td>$\pi, \Gamma \vdash \exists l_W : isWriter_{\text{reg}}(l_W, l_R) \land retv(l_R) = arg1(l_W)$</td>
</tr>
</tbody>
</table>

**Figure 25.** Derived Register Inference Rules
The rule BR\text{\textregistered} states that if a basic register \textit{reg} is sequentially-written, for every race-free read method call \textit{l}_R on \textit{reg}, there is a write method call \textit{l}_W on \textit{reg} that writes the same value that \textit{l}_R returns and \textit{l}_W is the last write method call that is executed before \textit{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 AR\text{\textregistered}’ states that for every read method call \textit{l}_R on an atomic register, if \textit{l}_W is the last write method call that is linearized before \textit{l}_R, then \textit{l}_W writes the same value that \textit{l}_R returns.

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

The rule BR\text{\textregistered}’ states that if a basic register \textit{reg} is accessed sequentially, for every read method call \textit{l}_R on \textit{reg}, there is a write method call \textit{l}_W on \textit{reg} that writes the same value that \textit{l}_R returns and \textit{l}_W is the last write method call that is executed before \textit{l}_R.

The rule TR\text{\textregistered} states that for every read method call \textit{l}_R on a thread-local register \textit{reg}, there is a write method call \textit{l}_W on \textit{reg} that writes the same value that \textit{l}_R returns and \textit{l}_W is the last write method call that is executed before \textit{l}_R.
The written value of the last successful write linearized before writes the same value that the written value of the last successful write linearized before.

The rule CASRegCASF states that for every read method call, there is a successful write method call or a successful cas method call. The written value writtenValue(ℓ) of a successful write method call ℓ is its first argument if it is a successful cas method call.

The rule CASRegCAST states that for every read method call ℓ on an atomic cas register, there is a successful write method call η that writes the same value that is linearized before η.

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

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

Figure 26. CAS Register Inference Rules.

Figure 27. Derived CAS Register Inference Rules

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

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

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

The rule CASRegCAST and the rule CASRegCASF state that a cas method call lC on an atomic cas register returns true if the written value of the last successful write linearized before lC is equal to the first argument of lC, 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 lR on an atomic cas register, the last successful write that is linearized before lR writes the same value that lR 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.

\[
\begin{align*}
isLock_o(l) & \iff \text{exec}(l) \land \text{obj}(l) = o \land \text{name}(l) = \text{lock} \\
isUnlock_o(l) & \iff \text{exec}(l) \land \text{obj}(l) = o \land \text{name}(l) = \text{unlock} \\
isRead_o(l) & \iff \text{exec}(l) \land \text{obj}(l) = o \land \text{name}(l) = \text{read} \\
isTryLock_o(l) & \iff \text{exec}(l) \land \text{obj}(l) = o \land \text{name}(l) = \text{tryLock} \\
isLock_o(l) \lor (isTryLock_o(l) \land \text{retv}(l) = \text{true}) \\
nUnlockBetween_o(l_l, l_u) & \iff \forall l_{u}' : (isUnlock_o(l_{u}') \land \text{thread}_X(l_l) = \text{thread}_X(l_{u}')) \Rightarrow (l_{u}' \prec l_l \lor l_u \preceq l_{u}')
\end{align*}
\]

\[
\begin{align*}
isOwnerRespecting(o) & \iff \forall l : isUnlock_o(l) \Rightarrow \exists l' : isTryLock_o(l') \land \text{thread}(l') = \text{thread}(l) \land l' < l \land \forall l'' : (isUnlock_o(l'') \land \text{thread}(l'') = \text{thread}(l)) \Rightarrow l'' < l' \lor l \leq l''
\end{align*}
\]

Figure 28. Preliminary definitions for Lock and TryLock Inference Rules.
Lock
\[ T_{\text{base}}(o) = \text{Lock} \]
\[ \pi, \Gamma \vdash \text{isOwnerRespecting}(o) \]
\[ \pi, \Gamma \vdash \text{isLock}_o(l_1) \quad \pi, \Gamma \vdash \text{isUnlock}_o(l_u) \]
\[ \pi, \Gamma \vdash l_1 \prec_o l_u \]
\[ \pi, \Gamma \vdash \exists \ell_{u_1}, \ell_{l_1} : \]
\[ \text{isUnlock}_o(\ell_{u_1}) \land \text{thread}(\ell_{u_1}) = \text{thread}(l_1) \land \]
\[ \text{isLock}_o(\ell_{l_1}) \land \text{thread}(\ell_{l_1}) = \text{thread}(l_u) \land \]
\[ l_{u_1} \prec_o \ell_{l_1} \]

TryLock
\[ T_{\text{base}}(o) = \text{TryLock} \]
\[ \pi, \Gamma \vdash \text{isOwnerRespecting}(o) \]
\[ \pi, \Gamma \vdash \text{isUnlock}_o(l_1) \quad \pi, \Gamma \vdash \text{isUnlock}_o(l_u) \]
\[ \pi, \Gamma \vdash l_1 \prec_o l_u \]
\[ \pi, \Gamma \vdash \exists \ell_{u_1}, \ell_{l_1} : \]
\[ \text{isUnlock}_o(\ell_{u_1}) \land \text{thread}(\ell_{u_1}) = \text{thread}(l_1) \land \]
\[ \text{isUnlock}_o(\ell_{l_1}) \land \text{thread}(\ell_{l_1}) = \text{thread}(l_u) \land \]
\[ l_{u_1} \prec_o \ell_{l_1} \]

LockReadL
\[ T_{\text{base}}(o) = \text{Lock} \]
\[ \pi, \Gamma \vdash \text{isOwnerRespecting}(o) \]
\[ \pi, \Gamma \vdash \text{isLock}_o(l_1) \]
\[ \pi, \Gamma \vdash \text{isRead}(l_r) \land \text{retv}(l_r) = \text{false} \]
\[ \pi, \Gamma \vdash l_r \prec_o l_u \]
\[ \pi, \Gamma \vdash \exists \ell_{l_1} : \]
\[ \text{isLock}_o(\ell_{l_1}) \land \text{thread}(\ell_{l_1}) = \text{thread}(l_1) \land \]
\[ l_{r_1} \prec_o \ell_{l_1} \]

TryLockReadL
\[ T_{\text{base}}(o) = \text{TryLock} \]
\[ \pi, \Gamma \vdash \text{isOwnerRespecting}(o) \]
\[ \pi, \Gamma \vdash \text{isUnlock}_o(l_1) \]
\[ \pi, \Gamma \vdash \text{isRead}(l_r) \land \text{retv}(l_r) = \text{false} \]
\[ \pi, \Gamma \vdash l_r \prec_o l_u \]
\[ \pi, \Gamma \vdash \exists \ell_{l_1} : \]
\[ \text{isUnlock}_o(\ell_{l_1}) \land \text{thread}(\ell_{l_1}) = \text{thread}(l_1) \land \]
\[ l_{r_1} \prec_o \ell_{l_1} \]

LockReadR
\[ T_{\text{base}}(o) = \text{Lock} \]
\[ \pi, \Gamma \vdash \text{isOwnerRespecting}(o) \]
\[ \pi, \Gamma \vdash \text{isLock}_o(l_1) \]
\[ \pi, \Gamma \vdash \text{isRead}(l_r) \land \text{retv}(l_r) = \text{false} \]
\[ \pi, \Gamma \vdash l_1 \prec_o l_r \]
\[ \pi, \Gamma \vdash \exists \ell_{u_1} : \]
\[ \text{isUnlock}_o(\ell_{u_1}) \land \text{thread}(\ell_{u_1}) = \text{thread}(l_1) \land \]
\[ l_{u_1} \prec_o \ell_{u_1} \]

TryLockReadR
\[ T_{\text{base}}(o) = \text{TryLock} \]
\[ \pi, \Gamma \vdash \text{isOwnerRespecting}(o) \]
\[ \pi, \Gamma \vdash \text{isUnlock}_o(l_1) \]
\[ \pi, \Gamma \vdash \text{isRead}(l_r) \land \text{retv}(l_r) = \text{false} \]
\[ \pi, \Gamma \vdash l_1 \prec_o l_r \]
\[ \pi, \Gamma \vdash \exists \ell_{u_1} : \]
\[ \text{isUnlock}_o(\ell_{u_1}) \land \text{thread}(\ell_{u_1}) = \text{thread}(l_1) \land \]
\[ l_{u_1} \prec_o \ell_{u_1} \]

LockReadM
\[ T_{\text{base}}(o) = \text{Lock} \]
\[ \pi, \Gamma \vdash \text{isOwnerRespecting}(o) \]
\[ \pi, \Gamma \vdash \text{isLock}_o(l_1) \quad \pi, \Gamma \vdash \text{isUnlock}_o(l_u) \]
\[ \pi, \Gamma \vdash \text{thread}(l_u) = \text{thread}(l_1) \]
\[ \pi, \Gamma \vdash \text{noUnlockBetween}_o(l_1, l_u) \]
\[ \pi, \Gamma \vdash \text{isRead}(l_r) \]
\[ \pi, \Gamma \vdash l_1 \prec_o l_{r_1} \quad \pi, \Gamma \vdash l_r \prec_o l_u \]
\[ \pi, \Gamma \vdash \text{retv}(l_r) = \text{true} \]

TryLockReadM
\[ T_{\text{base}}(o) = \text{TryLock} \]
\[ \pi, \Gamma \vdash \text{isOwnerRespecting}(o) \]
\[ \pi, \Gamma \vdash \text{isUnlock}_o(l_1) \quad \pi, \Gamma \vdash \text{isUnlock}_o(l_u) \]
\[ \pi, \Gamma \vdash \text{thread}(l_u) = \text{thread}(l_1) \]
\[ \pi, \Gamma \vdash \text{noUnlockBetween}_o(l_1, l_u) \]
\[ \pi, \Gamma \vdash \text{isRead}(l_r) \]
\[ \pi, \Gamma \vdash l_1 \prec_o l_r \quad \pi, \Gamma \vdash l_r \prec_o l_u \]
\[ \pi, \Gamma \vdash \text{retv}(l_r) = \text{true} \]

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.

\[
\begin{align*}
\text{SCounter} \\
\quad & T_{\text{base}}(o) = \text{SCounter} \\
& \quad \pi, \Gamma \vdash \text{obj}(l_1) = o \\
& \quad \pi, \Gamma \vdash \text{obj}(l_2) = o \land \text{name}(l_2) = \text{iaf} \\
& \quad \pi, \Gamma \vdash l_1 \prec_o l_2 \\
& \quad \pi, \Gamma \vdash \text{ret}(l_1) < \text{ret}(l_2)
\end{align*}
\]

**Figure 30.** SCounter Rules

\[
\begin{align*}
\text{SCounter’} \\
\quad & T_{\text{base}}(o) = \text{SCounter} \\
& \quad \pi, \Gamma \vdash \text{exec}(l_1) \land \text{obj}(l_1) = o \\
& \quad \pi, \Gamma \vdash \text{exec}(l_2) \land \text{obj}(l_2) = o \land \text{name}(l_2) = \text{iaf} \\
& \quad \pi, \Gamma \vdash \text{ret}(l_1) > \text{ret}(l_2) \\
& \quad \pi, \Gamma \vdash l_2 \prec_o l_1
\end{align*}
\]

**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 \( \text{isSequential}(o) \) if and only if every pair of method calls on \( o \) are ordered in the execution order.

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

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

The rule **BasicMapGet** states that if a basic map \( m \) is accessed sequentially, for every \( \text{get} \) method call \( l_g \) on \( m \) that does not return \( \bot \), there exists a \( \text{put} \) 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 \( \text{put} \) method call on \( m \) with the same key argument.

**Figure 32.** Set and Map Inference Rules

\[
\text{BasicSetContains} \\
\begin{align*}
\mathcal{T}_{\text{base}}(s) &= \text{BasicSet} \\
\pi, \Gamma \vdash \text{isSequential}(s) \\
\pi, \Gamma \vdash is\text{Contains}_s(l_c) \land \text{retv}(l_c) = \text{true} \\
&\quad \Rightarrow \pi, \Gamma \vdash \exists l_a: \text{isAdd}_s(l_a) \land arg1(l_a) = arg1(l_c) \land l_a < l_c
\end{align*}
\]

**BasicMapGet**

\[
\begin{align*}
\mathcal{T}_{\text{base}}(m) &= \text{BasicMap} \\
\pi, \Gamma \vdash \text{isSequential}(m) \\
\pi, \Gamma \vdash is\text{Get}_m(l_g) \land \text{retv}(l_g) \neq \bot \\
&\quad \Rightarrow \pi, \Gamma \vdash \exists l_p: \text{isPut}_m(l_p, l_g) \land arg2(l_p) = \text{retv}(l_g)
\end{align*}
\]

**BasicMapPut**

\[
\begin{align*}
\mathcal{T}_{\text{base}}(m) &= \text{BasicMap} \\
\pi, \Gamma \vdash \text{isSequential}(m) \\
\pi, \Gamma \vdash is\text{Put}_m(l_p) \\
\pi, \Gamma \vdash is\text{Put}_m(l_p) \\
&\quad \Rightarrow \pi, \Gamma \vdash arg2(l_p) = \text{retv}(l_g)
\end{align*}
\]

**Figure 33.** Derived Set and Map Inference Rules

\[
\text{BasicMapGet'} \\
\begin{align*}
\mathcal{T}_{\text{base}}(m) &= \text{BasicMap} \\
\pi, \Gamma \vdash \text{isSequential}(m) \\
\pi, \Gamma \vdash is\text{Get}_m(l_g) \\
\pi, \Gamma \vdash \neg \exists l_p: \text{isPut}_m(l_p) \land arg1(l_p) = arg1(l_g) \land l_p < l_g \\
&\quad \Rightarrow \pi, \Gamma \vdash \text{retv}(l_g) = \bot
\end{align*}
\]

**BasicMapPut’**

\[
\begin{align*}
\mathcal{T}_{\text{base}}(m) &= \text{BasicMap} \\
\pi, \Gamma \vdash \text{isSequential}(m) \\
\pi, \Gamma \vdash is\text{Get}_m(l_g) \\
\pi, \Gamma \vdash is\text{Put}_m(l_p) \\
&\quad \Rightarrow \pi, \Gamma \vdash \text{retv}(l_g) = \bot
\end{align*}
\]
The rule BasicMapPut states that if a basic map \( m \) is accessed sequentially, for every \( \text{get} \) method call \( l_g \) on \( m \), if \( l_p \) is the latest preceding \( \text{put} \) 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 \( \text{get} \) method call \( l_g \) on \( m \), if no \( \text{put} \) method call with the same key argument as \( l_g \) precedes \( l_g \), then \( l_g \) returns \( \bot \).

The rule BasicMapPut' states that if a basic map \( m \) is accessed sequentially and no \( \text{put} \) method call puts \( \bot \) in \( m \), every \( \text{get} \) method call that succeeds a \( \text{put} \) method call with the same key argument does not return \( \bot \).
12 Dekker Algorithm

In this section, we prove the mutual exclusion guarantee of the Dekker algorithm using the logic. We presented the Dekker algorithm, \( \pi_{\text{Dekker}} \), in Figure 1.

**Theorem 12.1** (Mutual Exclusion).
In every execution of the Dekker specification, at most one thread acquires the lock.

\( \forall X \in \mathbb{H}(\pi_{\text{Dekker}}) : (\text{ret} v_X(L_2) = \text{true}) \Rightarrow (\text{ret} v_X(L_1) = \text{false}) \).

**Proof.**

We show that

(1) \( X' \in \mathbb{H}(\pi_{\text{Dekker}}) \)

We show that

(2) \( (\text{ret} v_X(L_2) = \text{true}) \Rightarrow (\text{ret} v_X(L_1) = \text{false}) \)

By Definition 17 on [1], we have that there exists \( X, X', \sigma, \mathbb{L} \) such that

(3) \( X = (X, \sigma, \mathbb{L}) \in \mathbb{H}(\pi) \)

(4) \( X' = \sigma(X) \)

By Lemma 12.2, we have

(5) \( \pi_{\text{Dekker}} \models (\text{ret} v(L_2) = \text{true}) \Rightarrow (\text{ret} v(L_1) = \text{false}) \).

By the soundness theorem, Theorem 13.4, and Definition 13.3 on [5] and [3], we have

(6) \( X \models (\text{ret} v(L_2) = \text{true}) \Rightarrow (\text{ret} v(L_1) = \text{false}) \).

By the definition of \( \models \) (Figure 8) on [6], [3] and [4], we have

(7) \( (\text{ret} v_X(L_2) = \text{true}) \Rightarrow (\text{ret} v_X(L_1) = \text{false}) \).

**Lemma 12.2.**

\( \pi_{\text{Dekker}} \models (\text{ret} v(L_2) = \text{true}) \Rightarrow (\text{ret} v(L_1) = \text{false}) \).

**Proof.**

Let

\( \pi = \pi_{\text{Dekker}} \)

We show that

\( \pi \models (\text{ret} v(L_2) = \text{true}) \Rightarrow (\text{ret} v(L_1) = \text{false}) \).

Let

(8) \( \Gamma = (\text{ret} v(L_2) = \text{true}) \)

By rule CONCLUS, we have to show that

\( \pi, \Gamma \models \text{ret} v(L_1) = \text{false} \)

By rule Premise on [8], we have

(9) \( \pi, \Gamma \models \text{ret} v(L_2) = \text{true} \)

From \( \pi, \) we have

\( \text{cond}_e(L_2) = \text{true} \)

Thus,

(10) \( \pi, \Gamma \models \text{cond}_e(L_2) = \text{true} \)

By rule OCONTROL on [10], we have

(11) \( \pi, \Gamma \models \text{exec}(L_2) \)

From \( \pi, \) we have

(12) \( \text{name}_e(L_2) = \text{tryLock2} \)

(13) \( R_1 \in \text{Labels}(\text{tryLock2}) \)

From \( \pi, \) we have

\( \text{cond}_e(R_1) = \text{true} \)

Thus,

(14) \( \pi, \Gamma \models L_2 \text{cond}_e(R_1) = \text{true} \)

From \( \pi, \) we have

(15) \( \text{PreReturns}_e(R_1) = \emptyset \)

By rule ICONTROL on [11]-[15], we have

(16) \( \pi, \Gamma \models \text{exec}(L_2 \cdot R_1) \)

By rule In on [16], we have

(17) \( \pi, \Gamma \models \text{obj}(L_2 \cdot R_1) = f_1 \)

(18) \( \pi, \Gamma \models \text{name}(L_2 \cdot R_1) = \text{read} \)

(19) \( \pi, \Gamma \models \text{ret} v(L_2 \cdot R_1) = L_2 \cdot x_1 \)

Similarly, we have

(20) \( \pi, \Gamma \models \text{exec}(L_2 \cdot W_2) \)

(21) \( \pi, \Gamma \models \text{obj}(L_2 \cdot W_2) = f_2 \)

(22) \( \pi, \Gamma \models \text{name}(L_2 \cdot W_2) = \text{write} \)

(23) \( \pi, \Gamma \models \text{arg} 1(L_2 \cdot W_2) = 1 \)

From the definition of isRead on [16], [17] and [18] and rule CONJINTRO, we have

(24) \( \pi, \Gamma \models \text{isRead}_{f_1}(L_2 \cdot R_1) \)

From rule AREG on [24], we have

(25) \( \pi, \Gamma \models \exists \mathbb{L}' : \text{isWriter}_{f_1}(L_2 \cdot L_2 \cdot R_1) \land \text{arg} 1(L_2) = \text{ret} v(L_2 \cdot R_1) \)

Let

(26) \( \Gamma' = \Gamma ; \text{isWriter}_{f_1}(L_2 \cdot L_2 \cdot R_1) \land \text{arg} 1(L_2) = \text{ret} v(L_2 \cdot R_1) \)

where \( L_2 \) is fresh.

By rule Premise on [26], we have

(27) \( \pi, \Gamma' \models \text{isWriter}_{f_1}(L_2 \cdot L_2 \cdot R_1) \)

(28) \( \pi, \Gamma' \models \text{arg} 1(L_2) = \text{ret} v(L_2 \cdot R_1) \)

By rule In on [11], we have

(29) \( \pi, \Gamma \models \text{obj}(L_2) = \text{this} \)

(30) \( \pi, \Gamma \models \text{name}(L_2) = \text{tryLock2} \)

From \( \pi, \) we have

(31) \( \text{Returns}_e(\text{tryLock2}) = \{C_{21}, C_{2y}\} \)

By rule CALLER on [31], [11], [30], [31], we have

(32) \( \pi, \Gamma \models \text{exec}(L_2 \cdot C_{21}) \land \text{arg} 1(L_2 \cdot C_{21}) = \text{ret} v(L_2) \lor \text{exec}(L_2 \cdot C_{2y}) \land \text{arg} 1(L_2 \cdot C_{2y}) = \text{ret} v(L_2) \)

We apply rule DISJELIM to [32]:

Right:

Let

(33) \( \Gamma' = \Gamma ; \text{exec}(L_2 \cdot C_{2y}) \land \text{arg} 1(L_2 \cdot C_{2y}) = \text{ret} v(L_2) \)

By rule Premise on [33], we have

(34) \( \pi, \Gamma' \models \text{exec}(L_2 \cdot C_{2y}) \)
(35) \( \pi, \Gamma' \vdash \text{arg1}(L_2^C \cdot Q) = \text{retv}(L_2) \)
From \( \pi \), we have
(36) \( \text{arg1}(C \cdot Q) = \text{false} \)
By rule Id on [34], [36], we have
(37) \( \pi, \Gamma' \vdash \text{arg1}(L_2^C \cdot Q) = \text{false} \)
From rule ETrans and rule ESym on [35], and [37], we have
(38) \( \pi, \Gamma' \vdash \text{retv}(L_2) = \text{false} \)
By weakening (Lemma 13.2) on [33] [9], we have
(39) \( \pi, \Gamma' \vdash \text{retv}(L_2) = \text{true} \)
By rule NegElim on [38] and [39], we have
(40) \( \pi, \Gamma' \vdash \text{retv}(L_1) = \text{false} \)

Left:
Let
(41) \( \Gamma' = \Gamma; \)
\( \text{exec}(L_2^C \cdot Q) \wedge \text{arg1}(L_2^C \cdot Q) = \text{retv}(L_2) \)
By rule Premise on [41], we have
(42) \( \pi, \Gamma' \vdash \text{exec}(L_2^C \cdot Q) \)
(43) \( \pi, \Gamma' \vdash \text{arg1}(L_2^C \cdot Q) = \text{retv}(L_2) \)
From \( \pi \), we have
(44) \( \text{cond}_{\sigma}(C \cdot Q) = (x_1 = 0) \)
By rule IControl on [42] and [44] we have
(45) \( \pi, \Gamma' \vdash (L_2 \cdot x_1 = 0) \)
From [28], [19], [45], weakening (Lemma 13.2) and rule ETrans, we have
(46) \( \pi, \Gamma' \vdash \text{arg1}(i_0) = 0 \)
From the definition of isWriter on [27] and rule ConJELIM and rule ConJELIMR, we have
(47) \( \pi, \Gamma' \vdash \text{obj}(i_0) = f_1 \)
(48) \( \pi, \Gamma' \vdash \text{name}(i_0) = \text{write} \)
(49) \( \pi, \Gamma' \vdash \text{exec}(i_0) \)
(50) \( \pi, \Gamma' \vdash i_0 <_{f_1} L_2^C \cdot R_1 \)
(51) \( \pi, \Gamma' \vdash \forall L_2^C \cdot i_0 : \text{isWrite}_{f_1}(L_2^C) \Rightarrow \) \( l_0 \leq f_1 l_0 \) \( \lor \) \( L_2^C \cdot R_1 <_{f_1} L_2^C \cdot R_1 \)
From the definition of \( \pi \), we have
(52) \( \text{calls}_{\pi}(f_1, \text{write}) = \{W_1, W_0\} \)
From rule Ssc on [47], [48], [49] and [52], we have that for some fresh \( \varsigma \)
(53) \( \pi, \Gamma' \vdash i_0 = \varsigma \cdot W_1 \lor i_0 = \varsigma \cdot W_0 \)
We apply rule DisJELIM to [53]:

Left:
(54) \( \Gamma'' = \Gamma' \);
\( \pi, \Gamma'' \vdash \text{retv}(W_1) = 0 \)
From [49], [54], weakening (Lemma 13.2), we have
(55) \( \pi, \Gamma'' \vdash \text{exec}(\varsigma \cdot W_1) \)
From \( \pi \), we have
(56) \( \text{arg1}_{\pi}(W_1) = 1 \)
By rule Id on [54], [56], we have
(57) \( \pi, \Gamma'' \vdash \text{arg1}(\varsigma \cdot W_1) = 1 \)
From [54], [57], we have
(58) \( \pi, \Gamma'' \vdash \text{arg1}(i_0) = 1 \)
By weakening (Lemma 13.2) on [46], we have
(59) \( \pi, \Gamma'' \vdash \text{arg1}(i_0) = 0 \)
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) \( \pi, \Gamma'' \vdash \text{retv}(L_1) = \text{false} \)

Right:
(62) \( \Gamma'' = \Gamma' \);
\( l_0 = \varsigma \cdot W_0 \)
By rule Premise on [62], we have
(63) \( \pi, \Gamma'' \vdash l_0 = \varsigma \cdot W_0 \)
From \( \pi \), we have
(64) \( W_0 \in \text{Labels}_{\pi}(\text{init}) \)
By rule Call of [63] and [64] we have
(65) \( \pi, \Gamma'' \vdash \neg(\varsigma = e) \)
(66) \( \pi, \Gamma'' \vdash \text{exec}(\varsigma) \)
(67) \( \pi, \Gamma'' \vdash \text{obj}(\varsigma) = \text{this} \)
(68) \( \pi, \Gamma'' \vdash \text{name}(\varsigma) = \text{init} \)
From \( \pi \), we have
(69) \( \text{calls}_{\pi}(\text{this}, \text{init}) = (L_0) \)
By rule Ssc on [65]-[69] we have
(70) \( \pi, \Gamma'' \vdash \varsigma = L_0 \)
From [63] and [70], we have
(71) \( \pi, \Gamma'' \vdash l_0 = L_0 \cdot W_0 \)
From \( \pi \), we have
(72) \( \pi, \Gamma'' \vdash \text{cond}_{\pi}(L_1) = \text{true} \)
Thus,
(73) \( \pi, \Gamma'' \vdash L_1 \cdot \text{cond}_{\pi}(R_2) = \text{true} \)
By rule OControl on [72], we have
(74) \( \pi, \Gamma'' \vdash \text{exec}(L_1) \)
From \( \pi \), we have
(75) \( \text{name}_{\pi}(L_1) = \text{tryLock1} \)
(76) \( \text{R}_{\pi}(L_1) = \text{tryLock1} \)
From \( \pi \), we have
(77) \( \text{PreReturns}_{\pi}(R_2) = \emptyset \)
By rule IControl on [73]-[77], we have
(78) \( \pi, \Gamma'' \vdash \text{exec}(L_1 \cdot R_2) \)
From \( \pi \) we have
(79) \( \text{obj}_{\pi}(R_2) = f_2 \)
(80) \( \text{name}_{\pi}(R_2) = \text{read} \)
(81) \( \text{retv}_{\pi}(R_2) = x_2 \)
By rule Id on [78] and [79]-[81], and then rule ConJELIM and rule ConJELIMR, we have
(82) \( \pi, \Gamma'' \vdash \text{obj}(L_1 \cdot R_2) = f_2 \)
(83) \( \pi, \Gamma'' \vdash \text{name}(L_1 \cdot R_2) = \text{read} \)
(84) \( \pi, \Gamma'' \vdash \text{retv}(L_1 \cdot R_2) = L_1 \cdot x_2 \)
From the definition of isRead on [78], [82], [83] and rule ConJIntro, we have
(85) \( \pi, \Gamma'' \vdash \text{isRead}_{\pi}(L_1 \cdot R_2) \)
Similarly, we have that
\[(86) \pi, \Gamma' \vdash \text{exec}(L_1', W_1)\]
\[(87) \pi, \Gamma' \vdash \text{obj}(L_1', W_1) = f_1\]
\[(88) \pi, \Gamma' \vdash \text{name}(L_1', W_1) = \text{write}\]
\[(89) \pi, \Gamma' \vdash \text{arg1}(L_1', W_1) = 1\]
\[(90) \pi, \Gamma' \vdash \text{isWrite}_f(L_1', W_1)\]
By rule \text{UnivElim} on \[(91)\] and \[(90)\], we have
\[(91) \pi, \Gamma' \vdash L_1', W_1 \trianglerighteq f_1, L_0' W_1 \lor L_2' R_1 \triangleleft f_1, L_1' W_1\]
By rule \text{LSubs} on \[(91)\] and \[(71)\], we have
\[(92) \pi, \Gamma' \vdash \]
\[L_1', W_1 \trianglerighteq f_1, L_0' W_0_1 \lor L_2' R_1 \triangleleft f_1, L_1' W_1\]
From \(\pi\), we have
\[(93) L_0 \rightarrow_{\pi} L_1\]
By rule \text{LSubs} on \[(66)\] and \[(70)\], we have
\[(94) \pi, \Gamma' \vdash \text{exec}(L_0)\]
By rule \text{P2X} on \[(93)\], \[(94)\] and \[(73)\], we have
\[(95) \pi, \Gamma' \vdash L_0 < L_1\]
By rule \text{LSubs} on \[(49)\] and \[(71)\], we have
\[(96) \pi, \Gamma' \vdash \text{exec}(L_0' W_0_1)\]
By rule \text{OX2X} on \[(95)\], and \[(96)\], and \[(86)\], we have
\[(97) \pi, \Gamma' \vdash L_0' W_0_1 < L_1' W_1\]
By rule \text{I0} on \[(96)\], we have
\[(98) \pi, \Gamma' \vdash \text{obj}(L_0' W_0_1) = f_1\]
By rule \text{X2L} on \[(97)\], \[(98)\], and \[(87)\], we have
\[(99) \pi, \Gamma' \vdash L_0' W_0_1 \trianglerighteq f_1, L_1' W_1\]
By rule \text{LASM} on \[(99)\], and rule \text{ConjElimL}, we have
\[(100) \pi, \Gamma' \vdash \neg(L_1' W_1 \trianglerighteq f_1, L_0' W_0_1)\]
By rule \text{DisjSylL} on \[(92)\], \[(100)\], we have
\[(101) \pi, \Gamma' \vdash L_2' R_1 \triangleleft f_1, L_1' W_1\]
From \(\pi\), we have
\[(102) W_2 \rightarrow_{\pi} R_1\]
From rule \text{P2X} on \[(102)\], \[(20)\], \[(16)\], and weakening (Lemma 13.2), we have
\[(103) \pi, \Gamma' \vdash L_2' W_2 < L_2' R_1\]
From \(\pi\), we have
\[(104) W_1 \rightarrow_{\pi} R_2\]
From rule \text{P2X} on \[(104)\], \[(86)\] and \[(78)\], we have
\[(105) \pi, \Gamma' \vdash L_1' W_1 < L_1' R_2\]
From rule \text{XLT} on \[(103)\], \[(101)\] and \[(105)\], we have
\[(106) \pi, \Gamma' \vdash L_2' W_2 < L_1' R_2\]
From rule \text{X2L} on \[(106)\], \[(21)\] and \[(82)\], we have
\[(107) \pi, \Gamma' \vdash L_2' W_2 \trianglerighteq f_1, L_1' R_2\]
We show that
\[(108) \pi, \Gamma' \vdash \forall \ell_W:\]
\[\text{isWrite}_f(\ell_W) \Rightarrow \ell_W \trianglerighteq f_1, L_2' W_2 \lor L_1' R_2 \triangleleft f_1, \ell_W\]
Let
\[(109) \Gamma'' = \Gamma' ; \text{isWrite}_f(I''_W)\]
By rule \text{UnivIntro} and rule \text{CondIntro}, we have to show that
\[\pi, \Gamma'' \vdash I''_W \trianglerighteq f_1, L_2' W_2 \lor L_1' R_2 \triangleleft f_1, I''_W\]
By rule \text{Premise} on \[(109)\], we have
\[(110) \pi, \Gamma'' \vdash \text{isWrite}_f(I''_W)\]
From definition of \text{isWrite} on \[(110)\], we have
\[(111) \pi, \Gamma'' \vdash \]
\[\text{obj}(I''_W) = f_2 \land \text{name}(I''_W) = \text{write} \land \text{exec}(I''_W)\]
From the definition of \(\pi\), we have
\[(112) \text{calls}_\pi(f_2, \text{write}) = \{W_0_2, W_2\}\]
By rule \text{Src} on \[(111)\] and \[(112)\], we have that for some fresh \(\varsigma\),
\[(113) \pi, \Gamma'' \vdash I''_W \trianglerighteq \varsigma W_0_2 \lor I''_W \triangleleft \varsigma W_2\]
We apply rule \text{DisjElimL} on \[(113)\]:
\[\textbf{Left:}\]
\[(114) \Gamma'''' = \Gamma'' ; (I''_W \trianglerighteq \varsigma W_0_2)\]
By rule \text{Premise} on \[(114)\], we have
\[(115) \pi, \Gamma'''' \vdash \varsigma W_0_2\]
By rule \text{LSubs} on \[(111)\], \[(115)\] and weakening (Lemma 13.2), we have
\[(116) \pi, \Gamma'''' \vdash \text{exec}(\varsigma W_0_2)\]
From \(\pi\), we have
\[(117) W_0_2 \in \text{Labels}_\pi(\text{init})\]
By rule \text{Call} on \[(116)\] and \[(117)\], we have
\[(118) \pi, \Gamma'''' \vdash \neg(\varsigma = e)\]
\[(119) \pi, \Gamma'''' \vdash \text{exec}(\varsigma)\]
\[(120) \pi, \Gamma'''' \vdash \text{obj}(\varsigma) = \text{this}\]
\[(121) \pi, \Gamma'''' \vdash \text{name}(\varsigma) = \text{init}\]
From \(\pi\), we have
\[(122) \text{calls}_\pi(\text{this}, \text{init}) = \{L_0\}\]
By rule \text{Src} on \[(118)\]-\[(122)\], we have
\[(123) \pi, \Gamma'''' \vdash \varsigma = L_0\]
By rule \text{LSubs} on \[(115)\], \[(123)\], we have
\[(124) \pi, \Gamma'''' \vdash L_0' W_0_2\]
By rule \text{LSubs} on \[(111)\], \[(124)\], we have
\[(125) \pi, \Gamma'''' \vdash \text{obj}(L_0' W_0_2) = f_3\]
\[(126) \pi, \Gamma'''' \vdash \text{exec}(L_0' W_0_2)\]
From \(\pi\), we have
\[(127) L_0 \rightarrow_{\pi} L_2\]
By rule \text{P2X} on \[(127)\], \[(94)\] and \[(11)\], weakening (Lemma 13.2), we have
\[(128) \pi, \Gamma'''' \vdash L_0 < L_2\]
By rule \text{OX2X} on \[(128)\], and \[(126)\], and \[(20)\], we have
\[(129) \pi, \Gamma'''' \vdash L_0' W_0_2 < L_2' W_2\]
By rule \text{X2L} on \[(129)\], and \[(125)\], and \[(21)\], we have
\[(130) \pi, \Gamma'''' \vdash L_0' W_0_2 < f_1, L_2' W_2\]
By rule \text{DisjIntroL} on \[(130)\], we have
\[(131) \pi, \Gamma'''' \vdash L_0' W_0_2 \leq f_1, L_2' W_2 \lor L_1' R_2 \triangleleft f_1, L_0' W_0_2\]
By rule \text{LSubs} on \[(131)\] and \[(124)\], we have
\[(132) \pi, \Gamma'''' \vdash L_0' W_0_2 < L_2' W_2\]
\[ l'_w \leq_f L_2W_2 \lor \]
\[ L_1R_2 <_f l'_w \]

Right:

(133) \( \Gamma''' = \Gamma'''; (l'_w = \varsigma \ W_2) \)
By rule PREMISE on [133], we have
(134) \( \pi, \Gamma'''' \vdash l'_w = \varsigma \ W_2 \)
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) \( \pi, \Gamma'''' \vdash l'_w = L_2'W_2 \)
By rule DisjIntroR on [136], we have
(137) \( \pi, \Gamma'''' \vdash l'_w \leq_f L_2'W_2 \)
Thus, by rule DisjIntroL on [137], we have
\[ \pi, \Gamma'''' \vdash l'_w \leq_f L_2'W_2 \lor \]
\[ L_1'R_2 <_f l'_w \]

By rule ConjIntro and the definition of isWrite on [20]-[22] and weakening (Lemma 13.2), we have
(138) \( \pi, \Gamma'' \vdash isWrite_f(l_2'W_2) \)
By rule ConjIntro and the definition of isWriter on [138], [107], and [108], we have
(139) \( \pi, \Gamma'' \vdash isWriter_f(l_2'W_2, L_1'R_2) \)
By rule ConjIntro and the definition of isRead on [78], [82] and [83], we have
(140) \( \pi, \Gamma'' \vdash isRead_f(L_1'R_2) \)

From rule AReg’ on [140] and [139], we have
(141) \( \pi, \Gamma'' \vdash reto(L_1'R_2) = arg1(L_2'W_2) \)
By rule ETRANS and rule ESYM on [141], [84] and [23], we have
(142) \( \pi, \Gamma'' \vdash L_1'x_2 = 1 \)
By rule ZERO and rule ESubs on [142], we have
(143) \( \pi, \Gamma'' \vdash \neg(L_1'x_2 = 0) \)
From \( \pi \), we have that
(144) \( cond_\pi(C_y) = \neg(x_2 = 0) \)
(145) \( name_\pi(L_1) = tr\text{tryLock1} \)
(146) \( C_y \in Labels_\pi(tr\text{tryLock1}) \)
(147) \( Pre\text{Returns}_\pi(C_y) = \emptyset \)
From [144], we have
(148) \( L_1'\ cond_\pi(C_y) = \neg(L_1'x_2 = 0) \)
From [143] and [148], we have
(149) \( \pi, \Gamma'' \vdash L_1'\ cond_\pi(C_y) \)
By rule IControl on [73], [146], [145], [149], [147] we have
(150) \( \pi, \Gamma'' \vdash exec(L_1'\ C_y) \)
From \( \pi \), we have that
(151) \( C_y \in \text{Returns}_\pi(tr\text{tryLock1}) \)
(152) \( arg1_\pi(C_y) = \text{false} \)
By rule Id on [150] and [152], we have
(153) \( \pi, \Gamma'' \vdash arg1(L_1'\ C_y) = \text{false} \)
By rule Ret on [150], [151], we have
(154) \( \pi, \Gamma'' \vdash reto(L_1) = arg1(L_1'\ C_y) \)
By rule ETRANS and rule ESYM on [153], [154], we have
\( \pi, \Gamma'' \vdash reto(L_1) = \text{false} \)
\[ \square \]
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).
∀ π, Γ, A, A′, A′′:
(π, Γ; A; Γ′ ⊨ A′′) ⇒ (π, Γ; A′; Γ′ ⊨ A′′)

Lemma 13.2 (Weakening).
∀ π, Γ, A, A′:
(π, Γ ⊨ A) ⇒ (π, Γ; A′ ⊨ A)

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

Definition 13.3. A specification π models an assertion A if and only if every execution of π models A.
π ⊨ A iff ∀X ∈ [[π]] : X ⊨ A

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

Theorem 13.4 (Soundness).
∀ π, A: ((π, Γ ⊨ A) ∧ (π ⊨ Γ)) ⇒ (π ⊨ A).

See Section 15.2 for the proof.
14 Client Assertions

\[\Gamma_0 = \Gamma_1 \land \Gamma_2 \land \Gamma_3 \land \Gamma_4 \land \Gamma_5 \land \Gamma_6 \land \Gamma_7\]  
(136)

\[\Gamma_1 = \forall t: \text{Let } l = \text{initOf}(t) \land \text{isInit}(l) \land \text{thread}(l) = t\]  
(137)

\[\Gamma_2 = \forall \ell, \ell': (\text{isInit}(\ell) \land \text{isInit}(\ell') \land \text{thread}(\ell) = \text{thread}(\ell')) \Rightarrow \ell = \ell'\]  
(138)

\[\Gamma_3 = \forall \ell, \ell': (\text{isInit}(\ell) \land \text{exec}(\ell') \land \text{obj}(\ell') = \text{this} \land \text{thread}(\ell) = \text{thread}(\ell')) \Rightarrow \ell \leq \ell'\]  
(139)

\[\Gamma_4 = \forall l: \text{Let } l = \text{commitOf}(t) \land \text{isCommitted}(t) \Rightarrow (\text{isCommit}(l) \land \text{thread}(l) = t)\]  
(140)

\[\Gamma_5 = \forall \ell, \ell': (\text{isCommit}(\ell) \land \text{isCommit}(\ell') \land \text{thread}(\ell) = \text{thread}(\ell')) \Rightarrow \ell = \ell'\]  
(141)

\[\Gamma_6 = \forall \ell, \ell': (\text{exec}(\ell) \land \text{obj}(\ell) = \text{this} \land \text{isCommit}(\ell') \land \text{thread}(\ell) = \text{thread}(\ell')) \Rightarrow \ell \leq \ell'\]  
(142)

\[\Gamma_7 = \forall t: \text{isCommitted}(t) \lor \text{isAborted}(t)\]  
(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. \(\Gamma_1\): Every transaction is initialized. \(\Gamma_2\): Every transaction is initialized only once. \(\Gamma_3\): The initialization operation of each transaction is executed before its other operations. \(\Gamma_4\): If a transaction is committed, it executed the commit operation. \(\Gamma_5\): Every transaction executes the commit operation at most once. \(\Gamma_6\): The commit operation of each transaction is executed after its other operations. \(\Gamma_7\): 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) \( rEv(l) \lt_X iEv(l') \)

As \( X \) is a valid history, we have

4) \( iEv(l) \lt_X rEv(l) \)

5) \( iEv(l') \lt_X rEv(l') \)

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

6) \( iEv(l) \lt_X rEv(l') \)

From [6], we have

7) \( \neg (rEv(l') \lt_X iEv(l)) \)

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) \( rEv(l_1) \lt_X iEv(l_2) \)

From [2], we have

5) \( rEv(l_3) \lt_X iEv(l_4) \)

From [3], we have

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

From [6], we have

7) \( \neg (rEv(l_1) \lt_X iEv(l_2)) \)

From [7], we have

8) \( iEv(l_2) \lt_X rEv(l_3) \)

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

9) \( rEv(l_1) \lt_X iEv(l_4) \)

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 $\prec_X$.

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

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) $rEv(l) \prec_L iEv(l')$
From the well-formedness of the history $O$,
we have
(3) $iEv(l) \prec_L rEv(l)$
(4) $iEv(l') \prec_L rEv(l')$
From [3], [2] and [4], we have
(5) $iEv(l) \prec_L rEv(l')$
From [5], we have
(6) $\neg(rEv(l') \prec_L iEv(l))$
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
$l \prec_L l' \lor l' \prec_L l \lor l = l'$
Lemma 10.16:
Straightforward from the fact that $L$ is equivalent to $X$.
We have
1. $X \equiv L$
2. $L \in \text{SeqSpec}(o)$
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. $\text{obj}_L(l) = o$
7. $\text{obj}_L(l') = o$
From [1] on [4] and [5], we have
8. $l \in X$
9. $l' \in X$
From [1] on [6] and [7], we have
10. $\text{obj}_X(l) = o$
11. $\text{obj}_X(l') = o$

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).
∀π, A: ((π, Γ ⊢ A) ∧ (π ⊨ Γ)) ⇒ (π ⊨ A).

Proof:

HYPOTHESIS
1. π, Γ ⊢ A
2. X ⊨ Γ

DESIRED CONCLUSION
π ⊨ A

Let
3. π = (T, D, P)
4. D = d''
5. P = p₀, p₁ ⊃ ... ⊃ pₙ
6. X = (X, σ, L) ∈ [[π]]

By Definitions 13.3, we need to show that X ⊨ A

Let
7. X' = σ(X)

By definition [17] on [6] and [7], we have
8. X' ∈ Π(π)

By definition [13] on [6], we have
9. ∀o: Tbase(o) ∈ BT ⇒ X'|o ∈ Π₁(o)
10. ∀o: Tbase(o) ∈ LT ⇒ (X'|o, L(o)) ∈ Π₁(o)

By definition [87] on [10], we have
11. ∀i ∈ {0..n}: (X_i, σ) ∈ [[p_i]] ∧ X'' ∈ Interleave(X_0, ..., X_n) ∩ X = X₀ · X''

By definition [85] on [9], we have
12. ∀o: a ∈ Tbase(o) ∈ BT ⇒ (X'|o ∈ Sequential) ⇒ (X'|o ∈ SeqSpec(o))

By definition [87] on [10], we have
13. ∀o: a ∈ Tbase(o) ∈ LT ⇒ X'|o ≡ L(o) ∧ L(o) ∈ SeqSpec(o) ∧ X'|o ≡ L(o)

Induction on the derivation of [1]:

Case rule X2L:
By rule X2L on [1], we have that
14. Tbase(o) ∈ LT
15. π, Γ ⊢ l ⊨ l'
16. π, Γ ⊢ obj(l) = obj(l') = o
17. A = l ⊨ o l'

We show that X ⊨ A

That is

l ≺ L(σ(o)) l'

By the induction hypothesis on [16] and [17], and then [2], [6] and [7], we have
19. l ≺ X' l'
20. objX'(l) = objX'(l') = σ(o)

From [19] and [20], we have
21. l ≺ X'|σ(o) l'

By [15], we have
22. Tbase(σ(o)) ∈ LT
By [10] and [22], we have
23. (X'|σ(o), L(σ(o))) ∈ Π₁(o)

By Lemma 10.11 on [23] and [21], we have
l ≺ L(σ(o)) l'

Case rule Src:
We have that
24. A = √₀i=₁..c = c_i
25. π, Γ ⊢ exec(c')
26. π, Γ ⊢ obj(c') = θ
27. π, Γ ⊢ name(c') = n
28. Callsₙ(base(n)(θ), n) = {c_1}

We show that
A ⊨ √₀i=₁..c = c_i

that is

By the induction hypothesis on [25], [26], [27], and then [2], [6] and [7], we have
29. c' ∈ X'
30. objX'(c') = c'θ
31. nameX(c') = n

From [7] and [12] on [29], [30], [31], we have
32. c' ∈ X_i
33. objX_i(c') = c'θ
34. nameX_i(c') = n

By Lemma 15.2 on [11] and [32], we have
35. basename(objX_i(c')) = basename(c)
36. nameX_i(c') = name(c)

By the definition of basename and ′, we have
37. basename(θ) = basename(θ)

From [35] and [36], we have
38. basename(objX_i(c)) = basename(θ)

From [36] and [34], we have
39. name(c) = n

From the definition of callsₙ(base(n)(θ), n)
on [38] and [39], we have
(40) \( c \in \text{calls}_s(\text{basename}(\theta), n) \)
From [28] and [36], we have
\( \bigvee_{i=1..n} c = c_i \)

Case rule P2X:
We have that
(41) \( \mathcal{A} = \zeta' c_1 \leq \zeta' c_2 \)
(42) \( c_1 \rightarrow_\pi c_2 \)
(43) \( \pi, \Gamma \vdash \text{exec}(\zeta' c_1) \)
(44) \( \pi, \Gamma \vdash \text{exec}(\zeta' c_2) \)
We show that
\( X \models \zeta' c_1 < X' \zeta' c_2 \)
that is
\( \zeta' c_1 < X' \zeta' c_2 \)
By the induction hypothesis on [43], [44], and then [2], we have
(45) \( X \models \text{exec}(\zeta' c_1) \)
(46) \( X \models \text{exec}(\zeta' c_2) \)
that is
(47) \( \zeta' c_1 \in X' \)
(48) \( \zeta' c_2 \in X' \)
From Lemma 15.6 on [8], [47] and [48], we have
\( \zeta' c_1 < X' \zeta' c_2 \)

Case rule OX2IX:
We have that
(49) \( \mathcal{A} = c_1' c_3 < c_2' c_4 \)
(50) \( \pi, \Gamma \vdash c_1 < c_2 \)
(51) \( \pi, \Gamma \vdash \text{exec}(c_1' c_3) \)
(52) \( \pi, \Gamma \vdash \text{exec}(c_2' c_4) \)
We show that
\( X \models c_1' c_3 < c_2' c_4 \)
that is
\( c_1' c_3 < X' c_2' c_4 \)
By the induction hypothesis on [50], [51], [52], and then [2], we have
(53) \( X \models c_1 < c_2 \)
(54) \( X \models \text{exec}(c_1' c_3) \)
(55) \( X \models \text{exec}(c_2' c_4) \)
that is
(56) \( c_1 < X' c_2 \)
(57) \( c_1' c_3 \in X' \)
(58) \( c_2' c_4 \in X' \)
From [56], we have
(59) \( r\text{Ev}(c_1) <_{X'} r\text{Ev}(c_2) \)
From Lemma 15.7 on [8] and [57], we have
(60) \( r\text{Ev}(c_1' c_3) <_{X'} r\text{Ev}(c_1) \)
From Lemma 15.7 on [8], and [58], we have
(61) \( i\text{Ev}(c_2) <_{X'} i\text{Ev}(c_2' c_4) \)
From [60], [59] and [61], we have
(62) \( r\text{Ev}(c_1, c_3) <_{X'} i\text{Ev}(c_2' c_4) \)
From [62], we have
(63) \( c_1' c_3 <_{X'} c_2' c_4 \)

Case rule IControl:
We have that
(64) \( \mathcal{A} = \begin{cases} 
\text{exec}(c'c') 
\text{exec}(c) \land 
\bigvee_{i=1} c' = c_1 \land 
\sigma(c\text{cond}_\pi(c')) \land 
\bigwedge_{i=1..n} -\text{exec}(c_i) 
\end{cases} \)
(65) \( \text{Labels(name}_\pi(c)) = \{ \tau_r \} \)
(66) \( \text{PreReturns}_\pi(c') = \{ \tau_r \} \)
We show that
\( X \models \mathcal{A} \)
That is
\( c'c' \in X' \Rightarrow 
\begin{align*}
&c \in X' \land 
&\bigvee_{i=1} c' = c_1 \land 
&\sigma(c\text{cond}_\pi(c')) \land 
&\bigwedge_{i=1..n} -\text{exec}(c_i) 
\end{align*} \)
We first show that
\( c'c' \in X' \Rightarrow 
\begin{align*}
&c \in X' \land 
&\bigvee_{i=1} c' = c_1 \land 
&\sigma(c\text{cond}_\pi(c')) \land 
&\bigwedge_{i=1..n} -\text{exec}(c_i) 
\end{align*} \)
We assume that
(67) \( c'c' \in X' \)
We show that
\( c \in X' \land 
\begin{align*}
&\bigvee_{i=1} c' = c_1 \land 
&\sigma(c\text{cond}_\pi(c')) \land 
&\bigwedge_{i=1..n} -\text{exec}(c_i) 
\end{align*} \)
From [7] and [12] on [67], we have
\( \exists i \in \{0..n\} : 
\begin{align*}
&c \in X_i 
\begin{align*}
&\bigvee_{i=1} c' = c_i \land 
&\sigma(c\text{cond}_\pi(c')) \land 
&\bigwedge_{i=1..n} -\text{exec}(c_i) 
\end{align*} \)
From [7] and [12] and uniqueness of label \( c \) on [69], we have
(70) \( c \in X' \land 
\begin{align*}
&\bigvee_{i=1} c' = c_i \land 
&\sigma(c\text{cond}_\pi(c')) \land 
&\bigwedge_{i=1..n} -\text{exec}(c_i) 
\end{align*} \)
Now, we show that
\[ c \in X' \land \\
\forall c, c' = c_1 \land \\
\sigma(c' \mathord{\text{cond}}(c')) \land \\
\land_{i=1..n} \neg(c'_{ci} \in X') \]
\[ \Rightarrow c'c' \in X' \]

We assume that
\[ (71) c \in X' \land \\
(72) \forall c, c' = c_1 \land \\
(73) \sigma(c' \mathord{\text{cond}}(c')) \land \\
(74) \land_{i=1..n} \neg(c'_{ci} \in X') \]

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
\[ \forall i \in \{0..n\}: \\
(76) \land_{i=1..n} \neg(c'_{ci} \in X_i) \]

By Lemma 15.4 on [65], [66], [11], [75], [72], [73] and [76], we have
\[ (77) c'c' \in X_i \]

From [7] and [12] on [77], we have
\[ c'c' \in X' \]

Case rule OCONTROL:

Similar to rule ICONTROL using Lemma 15.5.

Case rule TSEQ:

We have that
\[ (78) A = l_1 < l_2 \lor l_2 < l_1 \lor l_1 = l_2 \\
(79) \pi, \Gamma \vdash \text{exec}(l_1) \\
(80) \pi, \Gamma \vdash \text{exec}(l_2) \\
(81) \pi, \Gamma \vdash \text{thread}(l_1) = \text{thread}(l_2) \\
(82) \pi, \Gamma \vdash \text{obj}(l_1) = \text{obj}(l_2) = \text{this} \lor \\
\neg\text{obj}(l_1) = \text{this} \land \neg\text{obj}(l_2) = \text{this} \]

We show that
\[ X \models l_1 < l_2 \lor l_2 < l_1 \lor l_1 = l_2 \]

that is
\[ l_1 <_X l_2 \lor l_2 <_X l_1 \lor l_1 = l_2 \]

By the induction hypothesis on [79], [80], [81], [82], and then [2], we have
\[ (83) X \models \text{exec}(l_1) \\
(84) X \models \text{exec}(l_2) \\
(85) X \models \text{thread}(l_1) = \text{thread}(l_2) \\
(86) X \models \text{obj}(l_1) = \text{obj}(l_2) = \text{this} \lor \\
\neg\text{obj}(l_1) = \text{this} \land \neg\text{obj}(l_2) = \text{this} \]

that is
\[ (87) l_1 \in X' \\
(88) l_2 \in X' \\
(89) \text{thread}(l_1) = \text{thread}(l_2) \]

\[ (90) \text{obj}(l_1) = \text{obj}(l_2) = \text{this} \lor \\
\neg\text{obj}(l_1) = \text{this} \land \neg\text{obj}(l_2) = \text{this} \]

By [11] and [12] on [87] and [88], we have
\[ \exists i, j \in \{0..n\}: \\
(91) l_i \in X_i \land (X_i, \sigma) \in [p_i] \\
(92) l_j \in X_j \land (X_j, \sigma) \in [p_j] \]

Case analysis on [90]:

Case
\[ (93) \text{obj}(l_1) = \text{obj}(l_2) = \text{this} \]

By Lemma 15.8 on [8], [87], [88], [93], we have

\[ (94) l_1 = c_1 \\
(95) l_2 = c_2 \]

By Lemma 15.10 on [91], [92], [94], [95], we have
\[ (96) \text{thread}(l_1) = T_i \\
(97) \text{thread}(l_2) = T_j \]

From [96], [97] and [89], we have
\[ (98) i = j \]

By Lemma 15.12 on [91], [92], and [94], [95], and [98], we have
\[ (99) l_i \prec X l_j \lor l_j \prec X l_i \lor l_1 = l_2 \]

Case
\[ (100) \neg\text{obj}(l_1) = \text{this} \land \\
\neg\text{obj}(l_2) = \text{this} \]

Similar to the previous case where lemmas 15.9, 15.11 and 15.13 are used.

Case rule TLOCAL:

We have that
\[ (101) A = \text{thread}(l_1) = \text{thread}(l_2) \\
(102) T (\text{basename}(\phi)) = \text{ThreadLocal st} \\
(103) \pi, \Gamma \vdash \text{exec}(l_1) \land \text{exec}(l_2) \\
(104) \pi, \Gamma \vdash \text{obj}(l_1) = \text{obj}(l_2) = \phi[u] \]

We show that
\[ X \models \text{thread}(l_1) = \text{thread}(l_2) \]

that is
\[ \text{thread}(l_1) = \text{thread}(l_2) \]

By the induction hypothesis on [104], and then [2], we have
\[ (105) X \models \text{exec}(l_1) \land \text{exec}(l_2) \\
(106) X \models \text{obj}(l_1) = \text{obj}(l_2) = \phi[u] \]

that is
\[ (107) \text{obj}(l_1) = \text{obj}(l_2) = \phi[u] \]

\[ (108) l_1 \in X' \\
(109) l_2 \in X' \]

From [107], we have
\[ (110) \text{basename}(\text{obj}(l_1)) = \phi \\
(111) \text{index}(\text{obj}(l_1)) = \sigma(u) \\
(112) \text{basename}(\text{obj}(l_2)) = \phi \\
(113) \text{index}(\text{obj}(l_2)) = \sigma(u) \]

From Lemma 15.14 on [3], [102], [8], [108] and
[110] we have
(114) \( \text{thread}_X(l_1) = \text{index}(\text{obj}_X(l_1)) \)
From Lemma 15.14 on [3], [102], [8], [109] and [112] we have
(115) \( \text{thread}_X(l_2) = \text{index}(\text{obj}_X(l_2)) \)
From [114] and [111] we have
(116) \( \text{thread}_X(l_1) = \sigma(u) \)
From [115] and [113] we have
(117) \( \text{thread}_X(l_2) = \sigma(u) \)
From [116] and [117] we have
(118) \( \text{thread}_X(l_1) = \text{thread}_X(l_2) \)

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

Case rule Caller:
We have that
(134) \( \mathcal{A} = \)
\( c't = \text{thread}_X(c) \land \)
\( c'x^* = \arg^*_X(c) \land \)
\( \bigvee_{i=1..n} (\text{exec}(c_i) \land \text{arg}_X(c_i) = \text{return}(c)) \)
(135) \( \pi, \Gamma, \Gamma \vdash \text{exec}(c) \)
(136) \( \pi, \Gamma, \Gamma \vdash \text{obj}(c) = \text{this} \)
(137) \( \pi, \Gamma, \Gamma \vdash \text{name}(c) = n \)
(138) \( \text{var}(n) = t \land \text{par}(n) = x \)
(139) \( \text{Return}_p(n) = (C_i) \)
We show that
\( X \models \mathcal{A} \)
that is
\( \sigma(c't) = \text{thread}_X(c) \land \)
\( \sigma(c'x^*) = \arg^*_X(c) \land \)
\( \bigvee_{i=1..n} (c_i \in X' \land \text{arg}_X(c_i) = \text{return}(c)) \)
By induction hypothesis on [135], [136] and [137], and then [2], [6] and [7], we have
(140) \( c \in X' \)
(141) \( \text{obj}_X(c) = \text{this} \)
(142) \( \text{name}_X(c) = n \)
From [7] on [140], [141] and [142], we have
(143) \( c \in X \)
(144) \( \text{obj}_X(c) = \text{this} \)
(145) \( \text{name}_X(c) = n \)
By Lemma 15.15 on [6], [138], [139], [143], [144], and [145], we have
(146) \( \sigma(c't) = \sigma(\text{thread}_X(c)) \land \)
(147) \( \sigma(c'x^*) = \sigma(\arg^*_X(c)) \land \)
(148) \( \bigvee_{i=1..n} (c_i \in X \land \sigma(\text{arg}_X(c_i)) = \sigma(\text{return}(c))) \)
From [7] on [146], [147], and [148], we have
\( \sigma(c't) = \text{thread}_X(c) \land \)
\( \sigma(c'x^*) = \arg^*_X(c) \land \)
\( \bigvee_{i=1..n} (c_i \in X' \land \text{arg}_X(c_i) = \text{return}(c)) \)

Case rule Ret:
We have that
\begin{align*}
(149) \ t_{par}(n) = t & \land \ par_1(n) = x \\
(150) \ c' \in Returns_{par}(n) & \\
(151) \ \sigma, \Gamma \vdash \text{exec}(c'c') & \\
(152) \ A = \\
\quad \text{exec}(c) & \land \\
\quad \text{obj}(c) = \text{this} & \land \ \text{name}(c) = n & \land \\
\quad \text{thread}(c) = c't & \land \ \text{arg}(c) = c'x' & \land \\
\quad \text{retv}(c) = \text{arg}1(c'c') & \\
\end{align*}

We show that
\begin{align*}
X & \vdash A \\
\text{that is} & \\
\quad c \in X' & \land \\
\quad \text{obj}_{X'}(c) = \text{this} & \land \ \text{name}_{X'}(c) = n & \land \\
\quad \text{thread}_{X'}(c) = \sigma(c't) & \land \\
\quad \text{arg}_{X'}(c) = \sigma(c'x') & \land \\
\quad \text{ret}_{X'}(c) = \text{arg}1_{X'}(c'c') & \\
\end{align*}

By induction hypothesis on [151],
and then [2], [6] and [7], we have
\begin{enumerate}
\item \( c'c' \in X' \)
\item From [7] and [153], we have
\begin{align*}
(154) \ c'c' \in X & \\
\end{align*}
\item From Lemma 15.17 on [6], [149], [150], and [154], we have
\begin{align*}
(155) \ c \in X & \\
(156) \ \text{obj}_{X'}(c) = \text{this} & \land \ \text{name}_{X'}(c) = n & \land \\
(157) \ \sigma(\text{thread}_{X'}(c)) = \sigma(c't) & \land \\
(158) \ \sigma(\text{arg}_{X'}(c)) = \sigma(c'x') & \land \\
(159) \ \sigma(\text{ret}_{X'}(c)) = \text{arg}1_{X'}(c'c') & \\
\end{align*}
\item From [7] on [155]-[159], we have
\begin{align*}
\quad c \in X' & \land \\
\quad \text{obj}_{X'}(c) = \text{this} & \land \ \text{name}_{X'}(c) = n & \land \\
\quad \text{thread}_{X'}(c) = \sigma(c't) & \land \\
\quad \text{arg}_{X'}(c) = \sigma(c'x') & \land \\
\quad \text{ret}_{X'}(c) = \text{arg}1_{X'}(c'c') & \\
\end{align*}

Case rule Callee:
Similar to rule Ret

Case rule XASYM:
We have that
\begin{align*}
(160) \ \sigma, \Gamma \vdash l \prec l' & \\
(161) \ A = \\
\quad \neg(l' \prec l) & \land \\
\quad \neg(l' \sim l) & \land \\
\quad \neg(l' = l) & \\
\end{align*}

We show that
\begin{align*}
X & \vdash A \\
\text{that is} & \\
\quad \neg(l' \prec_X l) & \land \\
\quad \neg(l' \sim_X l) & \land \\
\quad \neg(l' = l) & \\
\end{align*}

Straightforward from Lemma 10.1.

Case rule LTOTAL:

Case rule X2X:

Straightforward from Lemma 10.5.

Case rule LASYM:
We have that
\begin{align*}
(162) \ \sigma, \Gamma \vdash l \prec_o l' & \\
(163) \ A = \\
\quad \neg(l' \prec_o l) & \land \\
\quad \neg(l' = l) & \\
\end{align*}

We show that
\begin{align*}
X & \vdash A \\
\text{Let} & \\
(164) \ \sigma(o) = L(\sigma(o)) & \\
\text{We need to show that} & \\
\quad \neg(l' \prec_o l) & \land \\
\quad \neg(l' = l) & \\
\end{align*}

Straightforward from Lemma 10.13.

Case rule L2X:
We have that
\begin{align*}
(167) \ \sigma, \Gamma \vdash l \prec_o l' & \\
\end{align*}

Straightforward from Lemma 10.4.
(179) \( A = \text{exec}(l) \land \text{exec}(l') \land \\text{obj}(l) = \text{obj}(l') = o \)

We show that
\[ X \models A \]
that is
\[ l \in X' \land l' \in X' \land \text{obj}_{X'}(l) = \text{obj}_{X'}(l') = \sigma(o) \]

Let
\[ (180) O = \mathcal{L}(\sigma(o)) \]

By induction hypothesis on [178], and then [2], and [6], we have
\[ (181) l <_O l' \]

From [10] on [180], we have
\[ (182) (X'|\sigma(o), \mathcal{L}(\sigma(o))) \in \pi_{\mathcal{L}}(\sigma(o)) \]

By Lemma 10.16 on [182] and [181], we have
\[ l \in X' \land l' \in X' \land \text{obj}_{X'}(l) = \text{obj}_{X'}(l') = \sigma(o) \]

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
\[ \begin{align*}
183 \quad & \pi, \Gamma \vdash T \ll T' \\
184 \quad & \pi, \Gamma \vdash \text{exec}(l) \land \text{thread}(l) = T \\
185 \quad & \pi, \Gamma \vdash \text{exec}(l') \land \text{thread}(l') = T' \\
186 \quad & A = l < l' \lor l = l'
\end{align*} \]

We show that
\[ X \models A \]
that is
\[ l <_{X'} l' \]

By induction hypothesis on [183], [184], and [185], and then [2], [6] and [7], we have
\[ \begin{align*}
187 \quad & T \ll_{X'} T' \\
188 \quad & l \in X' \\
189 \quad & \text{thread}_{X'}(l) = T \\
190 \quad & l' \in X' \\
191 \quad & \text{thread}_{X'}(l') = T'
\end{align*} \]

From [189], we have
\[ l \in X'|T \]

From [189], we have
\[ l' \in X'|T' \]

From [187], we have
\[ \forall T, T': X'|T \ll_{H} X'|T' \]

From [194], [192] and [193], we have
\[ l <_{X'} l' \]

Case rule AREG:
We have that
\[ \begin{align*}
195 \quad & \mathcal{T}_{\text{base}}(\text{reg}) = \text{AtomicRegister} \\
196 \quad & \pi, \Gamma \vdash \text{isRead}_{\text{reg}}(l) \\
197 \quad & A = \exists W: \text{isWriter}_{\text{reg}}(l_W, l_R) \land \text{retv}(l_R) = \text{arg1}(l_W)
\end{align*} \]

Let
\[ (198) \text{reg}' = \sigma(\text{reg}) \]
\[ (199) \text{Reg} = \mathcal{L}(\text{reg}') \]

From [195] and [198], we have
\[ (200) \text{reg}' \in \text{AtomicRegister} \]

From [10] and [200], [199], we have
\[ (201) (X'|\text{reg}', \text{Reg}) \in \pi_{\mathcal{L}}(\text{reg}') \]

By the definition of isWriter on [197], we have
\[ (202) A = \exists W: \text{isWriter}_{\text{reg}}(l_W) \land l_W <_{\text{reg}} l_R \land \forall W': \text{isWriter}_{\text{reg}}(l_W') \Rightarrow (l_W' \preceq_{\text{reg}} l_W \lor l_R \preceq_{\text{reg}} l_W') \land \text{retv}(l_R) = \text{arg1}(l_W) \]

We show that
\[ X \models A \]
that is
\[ \exists W: \text{isXWrite}_{X', \text{reg}}(l_W) \land l_W <_{\text{Reg}} l_R \land \forall W': \text{isXWrite}_{X', \text{reg}}(l_W') \Rightarrow (l_W' \preceq_{\text{Reg}} l_W \lor l_R \preceq_{\text{Reg}} l_W') \land \text{retv}_{X'}(l_R) = \text{arg1}(l_W) \]

From [196], we have
\[ (203) \pi, \Gamma \vdash \text{exec}(l_R) \land \text{obj}(l_R) = \text{reg} \land \text{name}(l_R) = \text{read} \]

By induction hypothesis on [203], and then [2], [6] and [7], we have
\[ (204) l_R \in X' \land \text{obj}_{X'}(l_R) = \text{reg}' \land \text{name}_{X'}(l_R) = \text{read} \]

From the definition of isXRead on [204], we have
\[ (205) \text{isXRead}_{X', \text{reg}}(l_R) \]

By Lemma 10.22 on [200], [201] and [205], we have
\[ (206) \exists W: \text{isLWriter}_{X'|\text{reg}, \text{Reg}, \text{reg}}(l_W, l_R) \land \text{retv}_{X'|\text{reg}}(l_R) = \text{arg1}_{X'}(l_W) \]

From the definition of isLWriter on [206],
we have
\[ \exists l_w : isXWrite_{X'}(l_w) \land l_w \prec l_k \land \forall l_w' : isXWrite_{X'}(l_w') \Rightarrow \gamma(l_w \prec l_w') \land ret_{X'}(l_k) = arg_{X'}(l_w') \]

After simplification, we have
\[ \exists l_w : isXWrite_{X'}(l_w) \land l_w \prec l_k \land \forall l_w' : isXWrite_{X'}(l_w') \Rightarrow \gamma(l_w \prec l_w') \land ret_{X'}(l_k) = arg_{X'}(l_w') \]

By Lemma 10.25.

Case rule \texttt{BReg}:
Similar to rule \texttt{AReg} by Lemma 10.21.

Case rule \texttt{CASRegRead}:
By Lemma 10.24.

Case rule \texttt{CASRegCAST}:
By Lemma 10.25.

Case rule \texttt{CASRegCAST}:
By Lemma 10.25.

Case rule \texttt{Lock}:
We have that
\begin{align*}
(207) & \quad \theta_{base}(l_0) = \text{Lock} \\
(208) & \quad \pi, \Gamma \vdash isOwnerRespecting(l_0) \\
(209) & \quad \pi, \Gamma \vdash isLock_{l_0}(l_k) \\
(210) & \quad \pi, \Gamma \vdash isUnlock_{l_0}(l_u) \\
(211) & \quad L \vdash l_i \prec_{l_0} l_u \\
(212) & \quad \mathcal{A} = \exists l_u, l_i : \\
& \quad isUnlock_{l_0}(l_u) \land thread(l_u) = thread(l_i) \land isLock_{l_0}(l_i) \land thread(l_u) = thread(l_i) \land l_u \prec_{l_0} l_i
\end{align*}

Let
\begin{align*}
(213) & \quad l_o' = \sigma(l_0) \\
(214) & \quad L = L(l_o')
\end{align*}

We show that
\[ X \models \mathcal{A} \]
that is
\begin{align*}
(215) & \quad \exists l_u, l_i : \\
& \quad isXUnlock_{X'}(l_u) \land thread_{X'}(l_i) = thread_{X'}(l_u) \land isXLock_{X'}(l_i) \land thread_{X'}(l_i) = thread_{X'}(l_u) \land l_u \prec L l_i
\end{align*}

By induction hypothesis on [208]-[211], and then [2], [6] and [7], we have
\begin{align*}
(216) & \quad isXOwnerRespecting_{X'}(X') \land \\
(217) & \quad isXLock_{X'}(l_i) \land \\
(218) & \quad isXUnlock_{X'}(l_u) \land \\
(219) & \quad l_i \prec L l_u
\end{align*}

From [216]-[219], we have
\begin{align*}
(220) & \quad isXOwnerRespecting_{X'}(X'|l_o') \land \\
(221) & \quad isXLock_{X'}(l_i) \land \\
(222) & \quad isXUnlock_{X'}(l_u) \land \\
(223) & \quad l_i \prec L l_u
\end{align*}

From [207] and [213], we have
\begin{align*}
(224) & \quad l_o' \in \text{Lock}
\end{align*}

From Lemma 10.29 on [224], and [220]-[223], we have
\begin{align*}
(225) & \quad \exists l_u, l_i : \\
& \quad isXUnlock_{X'}(l_u) \land \\
(226) & \quad thread_{X'}(l_i) = thread_{X'}(l_u) \land \\
(227) & \quad isXLock_{X'}(l_i) \land \\
(228) & \quad thread_{X'}(l_i) = thread_{X'}(l_u) \land \\
(229) & \quad l_u \prec L l_i
\end{align*}

From [225]-[229], we have
\begin{align*}
(230) & \quad \exists l_u, l_i : \\
& \quad isXUnlock_{X'}(l_u) \land \\
(231) & \quad thread_{X'}(l_i) = thread_{X'}(l_u) \land \\
(232) & \quad isXLock_{X'}(l_i) \land \\
(233) & \quad thread_{X'}(l_i) = thread_{X'}(l_u) \land \\
(234) & \quad l_u \prec L l_i
\end{align*}

Case rule \texttt{LockReadL}:
Similar to the proof of rule \texttt{Lock}
using Lemma 10.30.

Case rule \texttt{LockReadR}:
Similar to the proof of rule \texttt{Lock}
using Lemma 10.31.

Case rule \texttt{TryLock}:
Similar to the proof of rule \texttt{Lock}
using Lemma 10.35.

Case rule \texttt{TryLockReadL}:
Similar to the proof of rule \texttt{Lock}
using Lemma 10.36.

Case rule \texttt{TryLockReadR}:
Similar to the proof of rule \texttt{Lock}
using Lemma 10.37.

Case rule \texttt{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.
\[ \forall p, X, \sigma, \varsigma, c': \]
\[ ((X, \sigma) \in \llbracket p \rrbracket \land \varsigma \land c' \in X) \]
\[ \Rightarrow \]
\[ (\text{obj}_X(\varsigma c') = \varsigma \text{obj}_\pi(c') \land \text{thread}_X(\varsigma c') = \varsigma \text{thread}_\pi(c') \land \text{name}_X(\varsigma c') = \text{name}_\pi(c') \land \text{arg1}_X(\varsigma c') = \varsigma \text{arg1}_\pi(c') \land \text{ret}_X(\varsigma c') = \varsigma \text{ret}_\pi(c')). \]

Proof.
Structural induction on \( p \):
(1) Case \( p = c \triangleright n_\tau(a') \)
   Straightforward form definition [1].
(2) Case \( p = p_1; p_2 \)
(3) Case \( p = \text{if} b p_1 \text{ else } p_2 \)
Straightforward form definition [12] and the induction hypothesis. □

Lemma 15.2.
\[ \forall p, X, \sigma, \varsigma, c': \]
\[ ((X, \sigma) \in \llbracket p \rrbracket \land \varsigma \land c' \in X) \]
\[ \Rightarrow \]
\[ \text{basename}(\text{obj}_X(\varsigma c')) = \text{obj}_\pi(c') \land \text{name}_X(\varsigma c') = \text{name}_\pi(c'). \]

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

Lemma 15.3.

Let
\[ \text{Labels}(\text{name}_\pi(c)) = \{c_i\} \]
\[ \text{PreReturns}_{\pi}(c') = \{c_r\} \]
\[ \forall p, X, \sigma, c, c': \]
\[ ((X, \sigma) \in \llbracket p \rrbracket \land c' \in X) \]
\[ \Rightarrow \]
\[ c \in X \land \forall_{c_r} c' = c_i \sigma(c' \text{cond}_\pi(c')) \land \mathcal{C}_{c_r} \neg(c' c_r \in X). \]

Proof.
Structural induction on \( p \):
(1) Case \( p = c \triangleright n_{\tau}(a') \)
   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 \( p = \text{if} b p_1 \text{ else } p_2 \)
Straightforward form definition [12] and
   the induction hypothesis. □

Lemma 15.4.

Let
Labels(name\(\pi\)(c)) = \{c\}
PreReturns\(\pi\)(c') = \{c\}'
\forall p, X, \sigma, c, c':
((X, \sigma) \in \llbracket p \rrbracket) \land
\ c \in X \land
\bigvee_{c_i} c' = c_i \land
\sigma(c' \text{cond}_{\pi}(c')) \land
\bigwedge_{c_i} \neg(c' c_i \in X))
\implies c' \in X.

Proof:

Structural induction on p:
(1) Case \(p = c \triangleright n.\text{u}:x\)
    Straightforward form definition [1]
(2) Case \(p = p_1; p_2\)

Lemma 15.5.
Let
\forall p, X, \sigma, c:
(X, \sigma) \in \llbracket p \rrbracket

\implies \sigma(\text{cond}_{\pi}(c))
\iff c \in X.

Proof:

Structural induction on p:
(1) Case \(p = c \triangleright n.\text{u}:x\)
    Straightforward form definition [1], [11], and the induction hypothesis.
(2) Case \(p = p_1; p_2\)
    Straightforward form definition [12] and the induction hypothesis.

Lemma 15.6.
\forall \pi, X, c_1, c_2:
X \in \mathbb{H}(\pi) \land
\ c_1 \in X \land
\ c_2 \in X \land
\ c_1 \xrightarrow{\pi} c_2
\implies c'_1 \ll_X c'_2.

Proof:

Case analysis on \(c_1 \xrightarrow{\pi} c_2\)
(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 \{c\}:
    \((c_i \rightarrow_n c_j) \land c'_i \in X' \land c'_j \in X') \implies c'_i \ll_{X'} c'_j \quad \square
Lemma 15.7.
\(\forall \pi, X, c, c':\)
\(X \in \Pi(\pi) \land c' \in X) \Rightarrow (iEv(c) \triangleleft_X iEv(c') \land rEv(c') \triangleleft_X rEv(c)).\)

Proof.
We have that
(1) \(X \in \Pi(\pi)\)
(2) \(c' \in X\)
We show that
(3) \(iEv(c) \triangleleft_X iEv(c')\)
(4) \(rEv(c') \triangleleft_X rEv(c)\)

From definition 17 and [13] on [1] and [2], we have
\(\exists X_i:\)
(3) \((X_i, \sigma) \in [p_i]\)
(4) \(c' \in X_i\)
(5) \(X_i \subseteq X\)
We show that
(6) \(iEv(c) \triangleleft_X iEv(c')\)
(7) \(rEv(c') \triangleleft_X rEv(c)\)

Structural induction on \(p:\)
(8) Case \(p = c \triangleright n.x(u): x\)
Straightforward form definition [1]
\(X = inv(c \triangleright n_x(u)) \cdot X' \cdot ret(c \triangleright x')\)
(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
(11) \(iEv(c) \triangleleft_X iEv(c')\)
(12) \(rEv(c') \triangleleft_X rEv(c)\)
\(\square\)

Lemma 15.8.
\(\forall \pi, X, \sigma, c:\)
\(X \in \Pi(\pi) \land l \in X \land\)
\(obj_X(l) = this \land\)
\(\Rightarrow \exists c : l = c.\)

Proof.
From definition 17 and [13], we have
\(\exists X_i:\)
(3) \((X_i, \sigma) \in [p_i]\)
(4) \(l \in X_i\)
(5) \(X_i \subseteq X\)

Straightforward form structural induction on \(p_i\) \(\square\)

Lemma 15.9.
\(\forall \pi, X, \sigma, c:\)
\(X \in \Pi(\pi) \land l \in X \land\)
\(\neg obj_X(l) = this \land\)
\(\Rightarrow \exists c, c' : l = c'.\)

Proof. Similar to Lemma 15.8. \(\square\)

Lemma 15.10.
\(\forall \pi, T, D, p_0, \ldots, p_n, X, \sigma, c:\)
\((\pi = (T, S, \mathcal{P}) \land\)
\(\mathcal{P} = p_0, p_1 || p_2 || \ldots || p_n) \land\)
\((X, \sigma) \in [p_i]\) \land
\(c \in X \land\)
\(\Rightarrow \text{thread}_X(c) = i.\)
Proof.

By structural induction on $p_i$, we have

1. $c \in \text{Labels}(p_i)$
2. $\text{thread}_X(c) = \text{thread}_\pi(c)$

Lemma 15.11.

$\forall \pi, \mathcal{T}, \mathcal{D}, \mathcal{P}, p_0, \ldots, p_n, X, \sigma, c$:

$\begin{align*}
(\pi &= (\mathcal{T}, \mathcal{D}, \mathcal{P}) \land \\
\mathcal{P} &= p_0, (p_1 || p_2 || \cdots || p_n) \land \\
(X, \sigma) &\in [p_i] \land \\
c'c' &\in X \land \\
\sigma(\text{thread}_X(c'c')) &= i.
\end{align*}$

Proof.

By structural induction on $p_i$, we have

$\exists n, \tau$:

1. $c' \in \text{Labels}(n)$
2. $\text{thread}_X(c'c') = c'\text{thread}_\pi(c')$
3. $\sigma(c'tpar_\pi(n)) = \sigma(\tau)$
4. $c \in X$

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. $\forall c \in \text{Labels}(p_i)$: $\text{thread}_\pi(c) = i$

From [1], [2] and [3], we have

4. $\text{thread}_X(c) = T_i$  \(\square\)

Lemma 15.12.

$\forall p, X, \sigma, c_1, c_2$:

$\begin{align*}
(X, \sigma) &\in [p] \land \\
c_1 &\in X \land \\
c_2 &\in X \land \\
\Rightarrow \quad c_1 <_X c_2 \lor c_2 <_X c_1 \lor c_1 = c_2.
\end{align*}$

Proof. Straightforward structural induction on $p$. \(\square\)

Lemma 15.13.

$\forall p, X, \sigma, c_1, c_2, c_3, c_4$:

$\begin{align*}
(X, \sigma) &\in [p] \land \\
c_1'c_2 &\in X \land \\
c_3'c_4 &\in X \land \\
\Rightarrow \quad c_1'c_2 <_X c_3'c_4 \lor c_3'c_4 <_X c_1'c_2 \lor c_1'c_2 = c_3'c_4.
\end{align*}$

Proof. Straightforward structural induction on $p$. \(\square\)

Lemma 15.14.

$\forall \pi, X, \phi, st$

$\begin{align*}
\pi &= (\mathcal{T}, \mathcal{D}, \mathcal{P}) \land \\
\mathcal{T}(\phi) &= \text{Threadlocal st} \land \\
X &\in \mathcal{H}(\pi) \land \\
l &\in X \land \\
\text{basename}(\text{obj}_X(l)) &= \phi \land \\
\Rightarrow \quad \text{thread}_X(l) = \text{index}(\text{obj}_X(l)).
\end{align*}$
Proof.

We have

1. \( \pi = (T, D, P) \)
2. \( T(\phi) = \text{Threadlocal st} \)
3. \( X \in \mathbb{X}(\pi) \)
4. \( I \in X \)
5. \( \text{basename}(\text{obj}_X(l)) = \phi \)

From definition 17 and 13 on [3] and [5], we have

\[ \exists X_i: \]
6. \( I \in X_i \)
7. \( (X_i, \sigma) \in [p_i] \)
8. \( \text{basename}(\text{obj}_X(l)) = \phi \)
9. \( X_i \in X \)

We show that

(10) \( \text{thread}_{X_i}(l) = \text{index}(\text{obj}_{X_i}(l)) \)

Structural induction on \( p_i \):

(11) Case \( p_i = c \rightarrow n(c', x) \)

Form definition [1], we have

(12) \( l = c'c' \)
(13) \( \text{index}(\text{object}_{X_i}(c', x)) = c'\text{index}_{X_i}(c') \)
(14) \( \text{thread}_{X_i}(c', x) = c'\text{thread}_{X_i}(c') \)

Lemma 15.15.

\[ \forall \pi, X, \sigma, L, c, n, t, x: \]
\[ (X, \sigma, L) \in [\pi] \]
\[ \text{tpar}_\pi(n) = t \land \text{par}_1\pi(n) = x \]
\[ \text{Returns}_{\pi}(n) = [\pi] \]
\[ c \in X \]
\[ \text{obj}_X(c) = \text{this} \]
\[ \text{name}_X(c) = n \]

\[ \Rightarrow \]
\[ \sigma(c't) = \text{thread}_{X}(c) \land \sigma(c'x') = \text{arg}^*_X(c) \land \]
\[ \bigwedge_{i=1..n} (c'c_i \in X \land \text{arg}^1_X(c'c_i) = \text{ret}v_X(c)). \]

Proof:

We have that

(1) \( (X, \sigma, L) \in [\pi] \)
(2) \( \text{tpar}_\pi(n) = t \land \text{par}_1\pi(n) = x \)
(3) \( \text{Returns}_{\pi}(n) = [\pi] \)
(4) \( c \in X \)
(5) \( \text{obj}_X(c) = \text{this} \)
(6) \( \text{name}_X(c) = n \)

We show that

\[ \sigma(c't) = \sigma(\text{thread}_{X}(c)) \land \sigma(c'x') = \sigma(\text{arg}^*_X(c)) \land \]
\[ \bigwedge_{i=1..n} (c'c_i \in X \land \]

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) \( \forall c' \in \text{Labels}(n): \text{thread}_{\pi}(c') = \text{tpar}_{\pi}(n) \)

From the well-formedness conditions, we have

The array access index to every thread-local object is the current thread identifier.

(16) \( \forall \phi, st, c': \)
\[ T(\phi) = \text{Threadlocal st} \land \]
\[ c' \in \text{Labels}(n) \Rightarrow \]
\[ \text{index}_{\pi}(c') = \text{tpar}_{\pi}(n) \]

From [13], [14], [15], [16], we have

(17) \( \text{thread}_{X_i}(l) = \text{index}(\text{obj}_{X_i}(l)) \)

(18) Case \( p_i = p' p'' \)

Straightforward form definition [11], the induction hypothesis and the uniqueness of label \( l \).

(19) Case \( p = \text{if } b \ p_i \text{ else } p_2 \)

Straightforward form definition [12] and the induction hypothesis.

From [10] and [9], we have

\[ \text{thread}_{X}(l) = \text{index}(\text{obj}_{X}(l)) \]

\[ \Box \]
\[ \sigma(\text{arg}1_{X_1}(c')\pi)) = \sigma(\text{ret}_X(c)) \]

Structural induction on \( p_i \):

(15) Case \( p_i = c' \Rightarrow n_\pi(u')x \)

From the Well-formedness condition of specifications that

Every branch of every method definition ends in a return statement.

we have

\( \exists c_i \in \{ \pi \}: \sigma(c'\text{cond}(c_i)) \)

The rest is straightforward form the following conditions of definition [1]

\( \forall c_i \in \{ \pi \} : \\
\quad c_i \in X' \Rightarrow \\
\quad (\sigma(c'\text{cond}(c_i)) \land \\
\quad \forall c_j \in \text{PreReturns}_\pi(c_i) \Rightarrow \neg c'c_j \in X') \\
\) and

\textbf{Lemma 15.16.}

\( \forall X, \sigma, c, n, \pi, u, x : \\
\quad (X, \sigma) \in \llbracket c' \Rightarrow n_\pi(u) : x \rrbracket \)

\( c', c'' \in \text{Returns}_\pi(n) \)

\( c'c' \in X \land c'c'' \in X \)

\( \Rightarrow \quad c' = c'' \).

\textbf{Proof.}

We have that

(1) \( (X, \sigma) \in \llbracket c' \Rightarrow n_\pi(u) : x \rrbracket \)

(2) \( c' \in \text{Returns}_\pi(n) \)

(3) \( c'' \in \text{Returns}_\pi(n) \)

(4) \( c'c' \in X \)

(5) \( c'c'' \in X \)

We show that

\( c' = c'' \)

We consider three cases

\textbf{Lemma 15.17.}

\( \forall \pi, X, \sigma, L, c, c', n, t, x : \\
\quad (X, \sigma, L) \in \llbracket \pi \rrbracket \)

\( tpar_\pi(n) = t \land \text{par1}_\pi(n) = x \)

\( c' \in \text{Returns}_\pi(n) \)

\( c'c' \in X \)

\( \Rightarrow \quad c \in X \land \\
\quad \text{obj}_X(c) = \text{this} \land \text{name}_X(c) = n \land \\
\quad \sigma(\text{thread}_X(c)) = \sigma(c't) \land \\
\quad \sigma(\text{arg}_X(c)) = \sigma(c'x') \land \\
\quad \sigma(\text{ret}_X(c)) = \sigma(\text{arg}_1 X(c'c')) \).

\( \forall c_r \in \{ \pi \} : \\
\quad c_r \in X' \Rightarrow \sigma(x') = \sigma(c'\text{arg}1_X(c_r)) \)

(16) Case \( p_i = p' p'' \)

Straightforward form definition [11], the induction hypothesis and the uniqueness of label \( c \).

(17) Case \( p = \text{if} b_{p_1} \text{else} p_2 \)

Straightforward form definition [12] and the induction hypothesis.

From [11] on [12], [13] and [14], we have

\( \sigma(c't) = \sigma(\text{thread}_X(c)) \land \\
\quad \sigma(c'x') = \sigma(\text{arg}_X(c)) \land \\
\quad \forall_{i=1..n} \\
\quad (c'c_i \in X \land \\
\quad \sigma(\text{arg}_1 X(c'c_i)) = \sigma(\text{ret}_X(c)) \) \]
Proof.

We have that
\[(1) \quad (X, \sigma, L) \in \llbracket \pi \rrbracket\]
\[(2) \quad t_{\pi}(n) = t \land par_{\pi}(n) = x\]
\[(3) \quad c' \in \text{Returns}_{\pi}(n)\]
\[(4) \quad c' c' \in X\]

We show that
\[(5) \quad c \in X \land obj_X(c) = \text{this} \land name_X(c) = n \land \sigma(\text{thread}_X(c)) = n\ |
\sigma(\text{arg}_X(c)) = \sigma(c' x) \land \sigma(\text{ret}_X(c)) = \sigma(\text{arg}_1 X(c' c'))\]

From definition 13 on \([1]\) and \([4]\), we have
\[(6) \quad (X_i, \sigma) \in \llbracket p_i \rrbracket\]
\[(7) \quad c' c' \in X_i\]
\[(8) \quad X_i \in X\]

We show that
\[(9) \quad c \in X_i \land obj_X(c) = \text{this} \land name_X(c) = n \land \sigma(\text{thread}_X(c)) = n\ |
\sigma(\text{arg}_X(c)) = \sigma(c' x) \land \sigma(\text{ret}_X(c)) = \sigma(\text{arg}_1 X(c' c'))\]

Structural induction on \(p_i\):
\[(13) \quad \text{Case } p_i = c \Rightarrow n, (u') : x\]
\[\quad \text{Straightforward form definition } [1] \text{ and Lemma 15.16.}\]
\[(14) \quad \text{Case } p_i = p' p''\]
\[\quad \text{Straightforward form definition } [11], \text{ the induction hypothesis and the uniqueness of label } c.\]
\[(15) \quad \text{Case } p = \text{if } b \text{ else } p_2\]
\[\quad \text{Straightforward form definition } [12] \text{ and the induction hypothesis.}\]

From \([11]\) on \([8]-[12]\), we have
\[c \in X \land obj_X(c) = \text{this} \land name_X(c) = n \land \sigma(\text{thread}_X(c)) = n \land \sigma(\text{arg}_X(c)) = \sigma(c' x) \land \sigma(\text{ret}_X(c)) = \sigma(\text{arg}_1 X(c' c'))\]
\(\square\)
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
\[(\pi, \Gamma \vdash isWriter_{reg}(l_W, l_R) \land isWriter_{reg}(l_W', l_R)) \Rightarrow (\pi, \Gamma \vdash l_W = l_W')\]

BReg’:
Derived from rule BReg and the following
\[\pi, \Gamma \vdash isSequential(reg) \Rightarrow \pi, \Gamma \vdash \forall \ell: (isRead_{reg}(\ell) \lor isWrite_{reg}(\ell)) \Rightarrow isRaceFree_{reg}(\ell)\]

TReg:
Derived from rule TLOCAL, rule TSEQ and rule BReg’.

CASRegREAD’:
Derived from rule CASRegREAD and the following
\[(\pi, \Gamma \vdash isWriter_{reg}(l_W, l_R) \land isWriter_{reg}(l_W', l_R)) \Rightarrow (\pi, \Gamma \vdash l_W = l_W')\]

SCounter’:
Derived from rule LTOTAL and rule SCounter.

BASICMapGET’:
Derived from rule BASICMapGET.

BASICMapPUT’:
Derived from rule BASICMapPUT.

DisjSyllL:
Derived from rule DisjELIM and rule NegELIM.

DisjSyllR:
Derived from rule DisjELIM and rule NegELIM.

CondElim’:
Derived from 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

\[\text{Inits}(X) = \{ l \mid l \in X \land \text{obj}_X(l) = \text{this} \land \text{name}_X(l) = \text{init}\}\] (144)

\[\text{Reads}(X) = \{ l \mid l \in X \land \text{obj}_X(l) = \text{this} \land \text{name}_X(l) = \text{read}\}\] (145)

\[\text{Writes}(X) = \{ l \mid l \in X \land \text{obj}_X(l) = \text{this} \land \text{name}_X(l) = \text{write}\}\] (146)

\[\text{Commits}(X) = \{ l \mid l \in X \land \text{obj}_X(l) = \text{this} \land \text{name}_X(l) = \text{commit}\}\] (147)

\[\text{Committed}(X) = \{ T \mid \exists l \in \text{Commits}(X) \land \text{thread}_X(l) = T \land \text{ret}_X(l) = \text{C}\}\] (148)

\[\text{Aborted}(X) = \{ T \mid \exists l \in X \land \text{obj}_X(l) = \text{this} \land \text{thread}_X(l) = T \land \text{ret}_X(l) = \text{A}\}\] (149)

Lemma 15.18.

\[\forall X, \sigma, c:\]
\[\begin{align*}
(X, \sigma) & \in \llbracket \text{trans} \rrbracket \land \\
c & \in X
\end{align*}\]
\[
\Rightarrow \\
(c \in \text{Inits}(X) \land c = IL_j) \lor \\
c \in \text{Reads}(X) \lor \\
c \in \text{Writes}(X) \lor \\
c \in \text{Commits}(X) \land c = CL_j) \land \\
(IL_j \leq c) \land \\
(CL_j \in X \Rightarrow c \leq CL_j)
\]

Proof:

Case \(j = 0\):

Case \(0 < j \leq n\):

Derived from Equation 82, induction on the structure of \(op\) and Equation 12. \(\square\)

Lemma 15.19.

\[\forall X, \sigma:\]

\[\begin{align*}
(X, \sigma) & \in \llbracket \text{trans} \rrbracket
\end{align*}\]
\[
\Rightarrow \\
\exists c:
\begin{align*}
c & \in X \land \text{obj}_X(c) = \text{this} \land \text{thread}_X(c) = j \land \\
(\text{ret}_X(c) = \text{C} \lor \text{ret}_X(c) = \text{A})
\end{align*}
\]

Proof:

Case \(j = 0\):

Derived from Equation 81, Equation 11, Equation 1 and the well-formedness condition

\(\forall c' \in \text{Returns}_\pi(\text{commit}) : \text{ret}_\pi(c') = \text{C} \lor \text{ret}_\pi(c') = \text{A}\).

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

\(\forall c' \in \text{Returns}_\pi(\text{commit}) : \text{ret}_\pi(c') = \text{C} \lor \text{ret}_\pi(c') = \text{A}\). \(\square\)

Lemma 15.20.

\[\forall X, \sigma, c, c':\]

\[\begin{align*}
(X, \sigma) & \in \llbracket \text{trans} \rrbracket \\
c & \in X \land \text{obj}_X(c) = \text{this} \land \text{thread}_X(c) = j \land \\
c' & \in X \land \text{obj}_X(c') = \text{this} \land \text{thread}_X(c') = j \land \\
(\text{ret}_X(c) = \text{C} \lor \text{ret}_X(c) = \text{C}) \lor (\text{ret}_X(c') = \text{A} \lor \text{ret}_X(c') = \text{A}) \Rightarrow \\
c = c'
\end{align*}\]

Proof:

Case \(j = 0\):

Derived from Equation 81, Equation 11, Equation 1 and the well-formedness conditions

\(\forall c \in \text{Returns}_\pi(\text{init}) : \text{arg}_1(c) = \text{ok}\)
∀c ∈ Returnsₚ(write): arg₁ₚ(c) ≠ C

and that in every execution of the transaction trans₀, all the write method calls return ok.

Case 0 < j ≤ n:
Derived from Equation 82, induction on the structure of op and Equation 12, Equation 1 and
the following well-formedness conditions
∀c ∈ Returnsₚ(init): arg₁ₚ(c) = ok
∀c ∈ Returnsₚ(read): arg₁ₚ(c) ≠ C
∀c ∈ Returnsₚ(write): arg₁ₚ(c) ≠ C
∀c ∈ Returnsₚ(commit): arg₁ₚ(c) = C ∨ arg₁ₚ(c) = A

Lemma 15.21.
∀π ∈ Πₘ: ∀X ∈ Ξ(π): ∀T ∈ Trans(X): Let l = commitOf(T): l ∈ Inits(X) ∧ threadₓ(l) = T
Proof. Derived from Equation 81, Equation 82, Equation 83, Equation 17, Equation 13, and Equation 11. □

Lemma 15.22.
∀π ∈ Πₘ: ∀X ∈ Ξ(π): ∀l, l′:
(l ∈ Inits(X) ∧ l′ ∈ Inits(X) ∧ threadₓ(l) = threadₓ(l′)) ⇒ l = l′

Lemma 15.23.
∀π ∈ Πₘ: ∀X ∈ Ξ(π): ∀l, l′:
(l ∈ Inits(X) ∧ l′ ∈ X ∧ objₓ(l′) = this ∧ threadₓ(l) = threadₓ(l′)) ⇒ l ≤ₓ l′

Lemma 15.24.
∀π ∈ Πₘ: ∀X ∈ Ξ(π): ∀T ∈ Trans(X)
Let l = commitOf(T):
T ∈ Committed(X) ⇒ (l ∈ Commits(X) ∧ threadₓ(l) = T)
Proof. Derived from Equation 84, Equation 17, Equation 13, Lemma 15.8, Lemma 15.18 and Lemma 15.10. □

Lemma 15.25.
∀π ∈ Πₘ: ∀X ∈ Ξ(π): ∀l, l′:
(l ∈ Commits(X) ∧ l′ ∈ Commits(X) ∧ threadₓ(l) = threadₓ(l′)) ⇒ l = l′
Proof. Derived from Equation 17, Equation 13, Lemma 15.8, Lemma 15.10 and Lemma 15.18. □

∀π ∈ Πₘ: ∀X ∈ Ξ(π): ∀l, l′:
(l ∈ X ∧ objₓ(l) = this ∧ l′ ∈ Commits(X) ∧ threadₓ(l) = threadₓ(l′)) ⇒ l ≤ₓ l′
Proof. Derived from Equation 17, Equation 13, Lemma 15.8, Lemma 15.10 and Lemma 15.18. □

Lemma 15.27.
∀π ∈ Πₘ: ∀X ∈ Ξ(π): ∀t: 0 ≤ t ≤ n
(t ∈ Committed(X) ∧ t ∈ ¬Aborted(X)) ∨ (t ∈ Aborted(X) ∧ t ∈ ¬Committed(X))

Lemma 14.1
∀π ∈ Πₘ: π ⊨ Γ₀.
Proof. Derived from Equations 144-149, Equations 136-142, the definition of ⊨ (Figure 8), Definition 13.3 and Lemmas 15.21-15.27. □