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 program logic and finally, the deduction of a lemma for the 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 program logic and finally, the deduction of a lemma for the example specification.

8.1 Algorithm Specification

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

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

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

\[
\begin{array}{|c|c|}
\hline
\mathcal{T}: & \mathcal{P}: \\
\hline
\text{lock: Lock} & L_1 \triangleright lock.lock() \\
\text{clock: SCounter} & L_2 \triangleright lock.lock() \\
\text{ver: BasicRegister} & C_1 \triangleright v_1 = clock.iaf() \quad \| \quad C_2 \triangleright v_2 = clock.iaf() \\
\hline
\end{array}
\]

Figure 12. Example Specification \(\pi\)

\[
\begin{align*}
\text{Figure 13. Structure Inference Rules.} \\
\end{align*}
\]

X2L

\[
\begin{align*}
\mathcal{T}(o) \in LT & \quad \pi, \Gamma \vdash obj(l) = obj(l') = o \quad \pi, \Gamma \vdash l < l' \\
\end{align*}
\]

XLTrans

\[
\begin{align*}
\pi, \Gamma \vdash l_1 < l_2 & \quad \pi, \Gamma \vdash l_2 < l_3 \quad \pi, \Gamma \vdash l_3 < l_4 \\
\end{align*}
\]

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

Figure 14. Basic inference rules.
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 that (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 that (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 that (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 $\mathcal{A}$.

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 that (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 that (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 that (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 $\text{<} \text{(execution order)}$ and $\prec \text{<} \text{(linearization order of linearizable object o)}$ and functions $\text{obj}$, $\text{name}$, $\text{thread}$, $\text{arg}1$ and $\text{retv}$. The assertion $\text{exec(l)}$ states that the method call labeled $l$ is executed. The assertion $l_1 \prec \prec l_2$ states that $l_1$ is executed before $l_2$. Any
concurrent execution on a linearizable object is equivalent to a correct sequential execution. The total order of method calls in the equivalent sequential execution is called the linearization order. For every linearizable object \( o \), the assertion \( l_1 \prec_o l_2 \) states that \( l_1 \) is before \( l_2 \) in the linearization order of \( o \). As \( \pi \) declares \textit{lock} and \textit{clock} as instances of linearizable types, the linearization orders of \textit{lock} and \textit{clock} are denoted by \( \prec_{\text{lock}} \) and \( \prec_{\text{clock}} \). We also use the equivalence relation on expressions and labels. The functions \( \text{obj}(l) \), \( \text{name}(l) \), \( \text{thread}(l) \), \( \text{arg1}(l) \), and \( \text{retv}(l) \) 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 \textit{CONTROL} states that a method call is executed if and only if its enclosing condition is satisfied. The introduction rule \textit{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 \textit{P2X} states the program-order-preservation property. If a method call \( l_i \) is ordered before a method call \( l_j \) in the program, and methods \( l_i \) and \( l_j \) are executed, then \( l_i \) is executed before \( l_j \). The rule \textit{Snc} intuitively states that every executed method originates from a call site in the specification. Let \( \text{Calls}_s(o,n) \) 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 \( \text{Calls}_s(o,n) \). For presentation purposes, this small example does not involve procedure calls and hence the rules \textit{Control}, \textit{In}, and \textit{Snc} are simplified.

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

**8.3 Deduction**

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

\[
\Gamma = \text{arg1}(R_1) < \text{arg1}(R_2)
\]

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

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

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

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

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

\[
\pi, \Gamma \vdash \text{exec}(R_1)
\]

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

\[
\pi, \Gamma \vdash \text{exec}(L_2)
\]

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

\[
\pi, \Gamma \vdash \text{exec}(R_2)
\]

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

From the rule \textit{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

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

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

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

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

\[
\pi, \Gamma \vdash C_1 \prec_{\text{clock}} C_2
\]
After skolemization of \(\ell\), the next step is to use rule P2X. From \(\pi\), we have
\[
L_1 \rightarrow_{\pi} C_1 \tag{40}
\]
\[
C_2 \rightarrow_{\pi} U_2 \tag{41}
\]
By the rule P2X on 40, 21 and 22, we have
\[
\pi, \Gamma \vdash L_1 \prec C_1 \tag{42}
\]
Similarly, by the rule P2X on 41, 26 and 28, we have
\[
\pi, \Gamma \vdash C_2 \prec U_2 \tag{43}
\]
By the rule XLTrans on 42, 39 and 43, we have
\[
\pi, \Gamma \vdash L_1 \prec U_2 \tag{44}
\]
By the rule In on 21, and the specification \(\pi\), we have
\[
\pi, \Gamma \vdash \text{obj}(L_1) = \text{lock} \tag{45}
\]
\[
\pi, \Gamma \vdash \text{name}(L_1) = \text{lock} \tag{46}
\]
\[
\pi, \Gamma \vdash \text{thread}(L_1) = T_1 \tag{47}
\]
Similarly, by the rule In on 28, and the specification \(\pi\), we have
\[
\pi, \Gamma \vdash \text{obj}(U_2) = \text{lock} \tag{48}
\]
\[
\pi, \Gamma \vdash \text{name}(U_2) = \text{lock} \tag{49}
\]
\[
\pi, \Gamma \vdash \text{thread}(U_2) = T_2 \tag{50}
\]
From rule X2L on 44, 45 and 48, we have
\[
\pi, \Gamma \vdash L_1 \prec U_2 \tag{51}
\]
Now, we use the rule LockUnlockPair. The proof of ownership respect can be done using the presented rules. For the sake of brevity, we skip the proof of ownership respect.
\[
\pi, \Gamma \vdash \text{isOwnerRespecting}(\text{lock}) \tag{52}
\]
From the definition of isLock on 21, 45 and 46, we have
\[
\pi, \Gamma \vdash \text{isLock}_{\text{lock}}(L_1) \tag{53}
\]
From the definition of isUnlock on 28, 48 and 49, we have
\[
\pi, \Gamma \vdash \text{isUnlock}_{\text{lock}}(U_2) \tag{54}
\]
By the rule LockUnlockPair on 52, 53, 54, and 51, and then substitution with 47 and 50, we have
\[
\pi, \Gamma \vdash \exists \ell_l, \ell_{l_{u}} : \text{unlock}_{\text{lock}}(\ell_{l_{u}}) \land \text{thread}(\ell_{l_{u}}) = T_1 \land \text{isLock}_{\text{lock}}(\ell_l) \land \text{thread}(\ell_l) = T_2 \land \ell_{l_{u}} \prec_{\text{lock}} \ell_l \tag{55}
\]
After skolemization of \(\ell_{l_{u}}\) and \(\ell_l\) with \(l_{u}\) and \(l_{l_{u}}\), we have
\[
\pi, \Gamma \vdash \text{unlock}_{\text{lock}}(l_{u}) \tag{56}
\]
\[
\pi, \Gamma \vdash \text{thread}(l_{u}) = T_1 \tag{57}
\]
\[
\pi, \Gamma \vdash \text{isLock}_{\text{lock}}(l_{l_{u}}) \tag{58}
\]
\[
\pi, \Gamma \vdash \text{thread}(l_{l_{u}}) = T_2 \tag{59}
\]
\[
\pi, \Gamma \vdash l_{u} \prec_{\text{lock}} l_{l_{u}} \tag{60}
\]
From the definition of isUnlock on 56, we have
\[
\pi, \Gamma \vdash \text{exec}(l_{u}) \tag{61}
\]
\[
\pi, \Gamma \vdash \text{obj}(l_{u}) = \text{lock} \tag{62}
\]
From \(\pi\), we have
\[
\text{Calls}_{\pi}(\text{lock}, \text{unlock}) = \{U_1, U_2\} \tag{63}
\]
By the rule Src on 61, 62, 63, and 64, we have
\[
\pi, \Gamma \vdash l_{u} = U_1 \lor l_{u} = U_2 \tag{64}
\]
Using negation introduction, from 50 and 57, we have
\[
\pi, \Gamma \vdash \neg(l_{u} = U_2) \tag{65}
\]
By disjunction syllogism on 65 and 66, we have
\[
\pi, \Gamma \vdash l_{u} = U_1 \tag{66}
\]
Similarly, using the rule Src, we can show that
\[
\pi, \Gamma \vdash l_{l_{u}} = L_2 \tag{67}
\]
By substitution of 67 and 68 to 60, we have
\[
\pi, \Gamma \vdash U_1 \prec_{\text{lock}} L_2 \tag{68}
\]
From \(\pi\), we have
\[
R_1 \rightarrow_{\pi} U_1 \tag{69}
\]
\[
L_2 \rightarrow_{\pi} R_2 \tag{70}
\]
By the rule P2X on 70, 23 and 24, we have
\[
\pi, \Gamma \vdash R_1 \prec U_1 \tag{71}
\]
By the rule P2X on 71, 25 and 27, we have
\[
\pi, \Gamma \vdash L_2 \prec R_2 \tag{72}
\]
By the rule XLTrans on 72, 69, and 73, we have
\[
\pi, \Gamma \vdash R_1 \prec R_2 \tag{73}
\]
9 Algorithm Description

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

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

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

\[
  c \triangleright \text{foreach } (i \in \text{set}) \quad (75)
\]

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

\[
  \text{slIter}(s, i) = c_i \triangleright b_i = \text{set.contains}(i), \quad \text{if } (b_i)
\]

\[
  \text{slIndexed}(s, i)
\]

where `slIndexed(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

\[
  \text{slIter}(s, 0), \quad (77)
\]
\[
  \text{slIter}(s, 1),
\]
\[
  \text{slIter}(s, 2),
\]
\[
  \ldots
\]
\[
  \text{slIter}(s, \text{max})
\]

where `max` is the maximum value stored in the set.

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

\[
  c \triangleright \text{foreach } ((i, v) \in \text{map}) \quad (78)
\]

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

\[
  \text{mlIter}(s, i) = c_i \triangleright v_i = \text{map.get}(i), \quad \text{if } (v_i \neq \perp)
\]

\[
  \text{mlIndexed}(s, i)
\]

where `mlIndexed(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

\[
  \text{mlIter}(s, 0), \quad (80)
\]
\[
  \text{mlIter}(s, 1),
\]
\[
  \text{mlIter}(s, 2),
\]
\[
  \ldots
\]
\[
  \text{mlIter}(s, \text{max})
\]

where `max` is the maximum key.

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

\[
  D_{TM} = \quad \text{def } \text{init}_i() s_0, r_0, \quad \text{def } \text{read}_i(i) s_1, r_1, \quad \text{def } \text{write}_i(i, v) s_2, r_2, \quad \text{def } \text{commit}_i() s_3, r_3,
\]

\[
  P_{TM} = \quad \text{trans}_{0, \text{trans}_1 || \text{trans}_2 || \ldots || \text{trans}_n}
\]

Transaction 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 `\text{init}_i()`, `\text{read}_i(i)`, `\text{write}_i(i, v)` and `\text{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 `\text{init}_i()` initializes the transaction `t`. The method `\text{read}_i(i)` returns the value of location `i` or `A` (if the transaction is aborted). The method `\text{write}_i(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 `\text{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 `\text{init}_i()`, `\text{write}_i(i, 0)` method calls for every location `i` and then `\text{commit}_i()`. Each transaction `\text{trans}_j`, `1 \leq j \leq n` starts with `\text{init}_i()` and then invokes a sequence of `\text{read}_j(i)` and `\text{write}_j(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 `\text{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:

\[
  \text{trans}_0 := I L_0 \triangleright \text{init}_0(); \quad (81)
\]
\[
  c_{00} \triangleright \text{write}_0(0, 0);
\]
\[
  c_{01} \triangleright \text{write}_0(1, 0);
\]
\[
  \ldots
\]
\[
  c_{0m} \triangleright \text{write}_0(m, 0);
\]
\[
  C L_0 \triangleright \text{commit}_0()
\]
Each transaction \( \text{trans}_j \) \( 1 \leq j \leq n \) is defined as follows:

\[
\text{trans}_j := \text{IL}_j \uparrow \text{init}_j(); \\
\text{op}_j := \text{c} \uparrow x = \text{read}_j(v_1,v_2) \;
\text{if} \ (\neg(x = A)) \end{cases}
\]
\[
\text{if} \ (\neg(x = A)) \end{cases}
\]

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

\( \forall c \in \text{Returns}_\pi(\text{init}): \pi_1(c) = \text{ok} \)

\( \forall c \in \text{Returns}_\pi(\text{read}): \pi_1(c) \neq \text{ok} \land \pi_1(c) \neq \text{C} \)

\( \forall c \in \text{Returns}_\pi(\text{write}): \pi_1(c) \neq \text{C} \)

\( \forall c \in \text{Returns}_\pi(\text{commit}): \pi_1(c) = \text{C} \lor \pi_1(c) = \text{A} \)

In addition, it is assumed that in every execution of the transaction \text{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 $l' < l''$ 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 $l_1 <_X l_2$, $l_2 <_X l_3$, 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 inv($l'$) $<_X$ inv($l''$) then $l <_X l''$.

**Lemma 10.7** (RX2X). For every execution history $X$ and method calls $l$, $l'$, and $l''$ if ret($l$) $<_X$ 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.

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 $o$, $H_B(o)$, is a set of histories that is constrained as follows:

$$H_B(o) \cap \text{Sequential} \subseteq SeqSpec(o)$$ (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 $o$, $H_L(o)$, is defined as the following set of execution and sequentialization pairs.

$$H_L(o) = \{(X, L) | X \equiv L \land L \in SeqSpec(o) \land \forall T \in X: \prec_{X|T} \subseteq \prec_L\}$$ (86)

Note that the notation 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 $o$, $H_L(o)$, is defined as the following set of execution and linearization pairs.

$$H_L(o) = \{(X, L) | X \equiv L \land L \in SeqSpec(o) \land \prec_X \subseteq \prec_L\}$$ (87)

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

We now present lemmas for serialization and linearization orders.

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

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

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

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

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

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

1 In this subsection, we use $\lor$ and $\exists$ as a notational convenience. $\forall l: p$ can be rewritten as $\forall (l \in Labels(X)) p(X)$ and $\exists l: p$ can be rewritten as $\exists (l \in Labels(X)) p(X)$. 24
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 \( l_1 \prec_X l_2 \prec_X l_3 \prec_X l_4 \), then \( l_1 \prec_X l_4 \)

See section 15.1.2 for proofs.

10.2.1 Register

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

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

\[
\begin{align*}
\text{isXRead}_{X,r}(l_r) & = l_r \in X \land \text{obj}_X(l_r) = r \land \text{name}_X(l_r) = \text{read} \\
\text{isXWrite}_{X,r}(l_w) & = l_w \in X \land \text{obj}_X(l_w) = r \land \text{name}_X(l_w) = \text{write} \\
\text{NoWriteBetween}_{X,r}(l_w,l_r) & = \forall l_w' : \text{isXWrite}_{X,r}(l_w') \Rightarrow (l_w' \leq_X l_w \lor l_r <_X l_w') \\
\text{isXWriter}_{X,r}(l_w,l_r) & = \text{isXWrite}_{X,r}(l_w) \land l_w <_X l_r \land \text{NoWriteBetween}_{X,r}(l_w,l_r) \\
\text{Legal}(r) & = \{ S \mid \forall l_r : \text{isXRead}_{S,r}(l_r) \Rightarrow \exists l_w : \text{isXWriter}_{S,r}(l_w,l_r) \land \text{ret}_{S}(l_r) = \text{arg1}_{S}(l_w) \} \\
\text{SeqSpec}(r) & = \{ S \mid S[r = S \land S \in \text{Sequential} \cap \text{Legal}(r) \}
\end{align*}
\]

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 read reads the value that the latest preceding write writes. Formally,

\[
\forall reg \in \text{BasicRegister} : \forall X \in \mathbb{H}_B(reg) : X \in \text{Sequential} \Rightarrow
\forall l_r : \text{isXRead}_{X,r}(l_r) \Rightarrow
\exists l_w : \text{isXWrite}_{X,r}(l_w,l_r) \land
\text{ret}_{X}(l_r) = \text{arg1}_{X}(l_w)
\]

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

\[
\begin{align*}
\text{isXRaceFree}_{X,r}(l) & = \forall l_w : \text{isXWrite}_{X,r}(l_w) \Rightarrow l_w \leq_X l \lor l <_X l_w \\
\text{isXSequentiallyWritten}_{r}(X) & = \forall l \in X : \text{isXWrite}_{X,r}(l) \Rightarrow \text{isXRaceFree}_{X,r}(l)
\end{align*}
\]

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

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

\[
\begin{align*}
\mathbb{H}_B(r) & = \{ X \mid X[o = X] \land \text{isXSequentiallyWritten}_{r}(X) \Rightarrow \forall l_r : \text{isXRead}_{X,r}(l_r) \land \text{isXRaceFree}_{X,r}(l_r) \Rightarrow
\exists l_w : \text{isXWriter}_{X,r}(l_w,l_r) \land
\text{ret}_{X}(l_r) = \text{arg1}_{X}(l_w) \}
\end{align*}
\]
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,

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

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

\[ \exists l_w: \text{isXWrite}_{X,r}(l_w, l_r) \land \]

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

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

Let AtomicRegister denote the type of atomic registers.

Let us define

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

\[ \text{isXWriter}_{X,L,r}(l_w, l_r) = \text{isXWrite}_{X,r}(l_w) \land l_w \preceq_L l_r \land \]

\[ L\text{NoWriteBetween}_{X,L,r}(l_w, l_r) \]

**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,

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

\[ \forall l_r: \text{isXRead}_{X,r}(l_r) \Rightarrow \]

\[ \exists l_w: \text{isXWrite}_{X,L,r}(l_w, l_r) \land \]

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

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.

Consider the following four concurrent threads.

- **T₁**: \( L_{11} \triangleright r_1.\text{write}(1) \)
- **T₂**: \( L_{21} \triangleright r_2.\text{write}(1) \)
- **T₃**: \( L_{31} \triangleright x_1 = r_1.\text{read}() \)
- **T₄**: \( L_{41} \triangleright y_2 = r_2.\text{read}() \)

If \( r_1 \) and \( r_2 \) are sequentially-consistent registers, there is an execution that results in the following values for the variables:

\( x_1 = 1, y_1 = 0, y_2 = 1 \) and \( x_2 = 0 \).

These values can be justified by the sequentialization order

(1) \( L_{r_1} = L_{42} \triangleright x_2 = r_2.\text{read}() \cdot L_{11} \triangleright r_1.\text{write}(1) \cdot L_{31} \triangleright x_1 = r_1.\text{read}() \)

for \( r_1 \) and the sequentialization order

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

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) \( L_{31} \preceq_X L_{32} \cdot (4) L_{41} \preceq_X L_{42} \). By XZL’ on 2, we have (5) \( L_{32} \preceq_X L_{41} \). By XXTRANS on 3, 5 and 4, we have (6) \( L_{31} \preceq_X L_{42} \). By XZL on 6, we have \( L_{31} \preceq_X r_2 \) 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 \( r.\text{cas}(v_1, v_2) \) updates the value of the register to \( v_2 \) and returns true if the current value of the register is \( v_1 \). It returns false otherwise.
A successful write is either a write method call or a successful cas method call. The written value of a successful write is its first argument, if it is a write method call or is its second argument, if it is a cas method call.

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

\[
\begin{align*}
\text{isXCAS}_{X,r}(l_W) & = l_W \in X \land \text{obj}_X(l_W) = r \land \text{name}_X(l_W) = \text{cas} \\
\text{isXWrite}_{X,r}(l_W) & = \text{isXWrite}(l_W) \lor (\text{isXCAS}(l_W) \land \text{retv}_X(l_W) = \text{true}) \\
\text{writtenValue}_X(l_W) & = \begin{cases} 
\text{arg}_1X(l_W) & \text{if name}_X(l_W) = \text{write} \\
\text{arg}_2X(l_W) & \text{if name}_X(l_W) = \text{cas} 
\end{cases} \\
\text{LNoWriteBetween}_{X,L,r}(l_W,l_R) & = \forall l_W': \text{isXWrite}_{X,r}(l_W') \Rightarrow (l_W' \preceq_L l_W \lor l_R \prec_L l_W') \\
\text{isLCWriter}_{X,L,r}(l_W,l_R) & = \text{isXWrite}_{X,r}(l_W) \land l_W \prec_L l_R \\
\end{align*}
\]

**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,

\[
\forall r \in \text{AtomicCASRegister} : \forall (X, L) \in \mathbb{H}_L(r) : \\
\forall l_R : \text{isXRead}_{X,r}(l_R) \Rightarrow \\
\exists l_W : \text{isLCWriter}_{X,L,r}(l_W,l_R) \land \\
\text{retv}_X(l_R) = \text{arg}_1X(l_W)
\]

**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,

\[
\forall \text{reg} \in \text{AtomicCASRegister} : \forall (X, Reg) \in \mathbb{H}_L(\text{reg}) : \\
\forall l_C, l_W : \\
\text{isXCAS}_{X,\text{reg}}(l_C) \land \\
\text{isLCWriter}_{X,\text{Reg,reg}}(l_W,l_R) \\
\Rightarrow \\
(\text{writtenValue}_X(l_W) = \text{arg}_1X(l_C) \Rightarrow \text{retv}_X(l_C) = \text{true}) \land \\
(\neg(\text{writtenValue}_X(l_W) = \text{arg}_1X(l_C)) \Rightarrow \text{retv}_X(l_C) = \text{false})
\]

**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.

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

(109)

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

(110)

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

(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.

\[
isXOwnerRespecting_{o}(X) = \begin{cases} \forall l: \text{isXUnlock}_{X,o}(l) \Rightarrow \\ \exists l': \text{isXLock}_{X,o}(l') \land \\ \text{thread}_X(l') = \text{thread}_X(l) \land \\ l' \prec_X l \land \\ \forall l'': (\text{isXUnlock}_{X,o}(l'') \land \text{thread}_X(l'') = \text{thread}_X(l)) \Rightarrow (l'' \prec_X l' \lor l \leq_X l'') \end{cases}
\]

(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

\text{isXLock}_{X,o}(l_1) \land

\text{isXUnlock}_{X,o}(l_2) \land

l_1 \leq_L l_2) \Rightarrow

\exists l_{u1}, l_{u2}:

\text{isXUnlock}_{X,o}(l_{u1}) \land \text{thread}_X(l_{u1}) = \text{thread}_X(l_1) \land

\text{isXLock}_{X,o}(l_{u2}) \land \text{thread}_X(l_{u2}) = \text{thread}_X(l_2) \land

l_{u1} \leq_L l_{u2}
\]

(113)

**Lemma 10.30 (LockRead).** 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,

\[
\forall o \in \text{Lock}: \forall (X,L) \in \mathbb{H}_L(o): \forall l_{r1}, l_{r2}:
\]

\[
(isXOwnerRespecting_{o}(X) \land

\text{isXRead}_{X,o}(l_{r2}) \land \text{ret}_X(l_{r2}) = \text{false} \land

\text{isXUnlock}_{X,o}(l_{u1}) \land

l_{r2} \leq_L l_{u1}) \Rightarrow

\exists l_{r1}:

\text{isXLock}_{X,o}(l_{r1}) \land \text{thread}_X(l_{r1}) = \text{thread}_X(l_{u1}) \land

l_{r2} \leq_L l_{r1}
\]

(114)
Abstract Try-lock. A try-lock is an object that encapsulates an abstract state, acquired $\mathbb{A}$ or released $\mathbb{R}$, and in addition to lock, unlock and read methods, it supports the trylock method. If the state of the lock is $\mathbb{R}$, l.trylock() changes it to $\mathbb{A}$ and returns true. Otherwise, it returns false.

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

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

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

Let TryLock denote the type of try-locks.

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

\[
isXTryLock_{X,o}(l) = \begin{cases} & l \in X \land obj_X(l) = o \land name_X(l) = \text{tryLock} \\
& \text{isXTLock}_{X,o}(l) = \text{isXLock}_{X,o}(l) \lor (\text{isXTryLock}_{X,o}(l) \land \text{retv}_X(l) = \text{true}) \end{cases}
\]

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.

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

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

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

\[
\forall o \in \text{TryLock} : \forall (X, L) \in \mathcal{H}_L(o) : \forall l_{t_1}, l_{t_2}:
\]

\[
(\text{isXTOwnerRespecting}_o(X) \land \\
\text{isXTLock}_{X, o}(l_{t_1}) \land \\
\text{isXUnlock}_{X, o}(l_{t_2}) \land \\
l_{t_1} <_L l_{t_2}) \Rightarrow \\
\exists l_{t_{u_1}}, l_{t_{u_2}}:
\]

\[
\text{isXUnlock}_{X, o}(l_{t_{u_1}}) \land \text{thread}_X(l_{t_1}) = \text{thread}_X(l_{t_{u_1}}) \land \\
\text{isXTLock}_{X, o}(l_{t_{u_2}}) \land \text{thread}_X(l_{t_{u_2}}) = \text{thread}_X(l_{t_2}) \land \\
l_{t_{u_1}} <_L l_{t_{u_2}}
\]

**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,

\[
\forall o \in \text{TryLock} : \forall (X, L) \in \mathcal{H}_L(o) : \forall l_{u_1}, l_{u_2}:
\]

\[
(\text{isXTOwnerRespecting}_o(X) \land \\
\text{isXRead}_{X, o}(l_{u_2}) \land \text{ret}_X(l_{u_2}) = \text{false} \land \\
\text{isXUnlock}_{X, o}(l_{u_1}) \land \\
l_{u_2} <_L l_{u_1}) \Rightarrow \\
\exists l_{t_1}:
\]

\[
\text{isXTLock}_{X, o}(l_{t_1}) \land \text{thread}_X(l_{t_1}) = \text{thread}_X(l_{t_{u_1}}) \land \\
l_{t_2} <_L l_{t_1}
\]

**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,

\[
\forall o \in \text{TryLock} : \forall (X, L) \in \mathcal{H}_L(o) : \forall l_{t_1}, l_{t_2}:
\]

\[
(\text{isXTOwnerRespecting}_o(X) \land \\
\text{isXTLock}_{X, o}(l_{t_1}) \land \\
\text{isXRead}_{X, o}(l_{t_2}) \land \text{ret}_X(l_{t_2}) = \text{false} \land \\
l_{t_1} <_L l_{t_2}) \Rightarrow \\
\exists l_{u_1}:
\]

\[
\text{isXUnlock}_{X, o}(l_{u_1}) \land \text{thread}_X(l_{t_1}) = \text{thread}_X(l_{u_1}) \land \\
l_{u_1} <_L l_{t_2}
\]
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,

\[
\forall o \in \text{TryLock}: \forall (X, L) \in \mathbb{H}_L(o): \forall l_{11}, l_{u1}, l_{r2}:
\]

\[
\begin{array}{l}
\text{(isXOwnerRespecting}_o(X) \land \\
isXTLock\_X,o(l_{11}) \land \\
isXUnlock\_X,o(l_{u1}) \land \\
\text{thread}_X(l_{11}) = \text{thread}_X(l_{u1}) \land \\
\forall l'_{u1}: (\text{isXUnlock}_X,o(l'_{u1}) \land \text{thread}_X(l_{11}) = \text{thread}_X(l'_{u1})) \Rightarrow (l'_{u1} \prec X l_{11} \lor l_{u1} \preceq X l'_{u1}) \\
isXRead\_X,o(l_{r2}) \land \\
l_{11} \prec L l_{r2} \land l_{r2} \preceq L l_{u1} \Rightarrow \\
\text{ret}_X(l_{r2}) = true
\end{array}
\] (123)

10.2.5 Sequence-lock

Abstract seq-lock. A seq-lock \( l \) is an object that encapsulates a number and an abstract state, acquired \( A \) or released \( R \). It supports the read, compareAndLock and incAndUnlock methods. The method call \( l.\text{read}() \) returns the pair of the encapsulated number and true if the state of lock is \( A \) and false otherwise. The method call \( l.\text{compareAndLock}(n) \) 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 \( l.\text{incAndUnlock}() \) 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 \( c.\text{read}() \) returns the current value of \( c \). The method call \( c.\text{iaf}() \) 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 if method call is smaller than the return value of the if method call. Formally,

\[ \forall c \in SCounter: \forall (X, C) \in \mathbb{H}_c(l) : \forall l, l': \]
\[ l \in X \land l' \in X \land name_X(l') = if \land l <_c l' \]
\[ \Rightarrow retv_X(l) < retv_X(l') \]  

(124)

### 10.2.7 Set

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

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

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

Let BasicSet denote the type of basic sets.

Let us define

\[\text{isXContains}_{X,s}(l) = \]
\[ l \in X \land obj_X(l) = s \land name_X(l) = \text{contains} \]

\[\text{isXAdd}_{X,s}(l) = \]
\[ l \in X \land obj_X(l) = s \land name_X(l) = \text{add} \]  

(125)

(126)

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

\[ \forall s \in \text{BasicSet}: \forall X \in \mathbb{H}_R(s) : X \in \text{Sequential} \Rightarrow \]
\[ \forall l_c: \text{isXContains}_{X,s}(l_c) \land retv_X(l_c) = true \Rightarrow \]
\[ \exists l_a: \text{isXAdd}_{X,s}(l_a) \land ar1(l_a) = ar1(l_c) \land l_a <_X l_c \]  

(127)

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

\[ \forall s \in \text{BasicSet}: \forall X \in \mathbb{H}_R(s) : X \in \text{Sequential} \Rightarrow \]
\[ \forall l_c, l_a: \]
\[ \text{isXContains}_{X,s}(l_c) \land \]
\[ \text{isXAdd}_{X,s}(l_a) \land \]
\[ ar1(l_a) = ar1(l_c) \land l_a <_X l_c \]
\[ \Rightarrow \]
\[ retv_X(l_c) = true \]  

(128)

### 10.2.8 Map

A map \( m \) is an object that represents a mapping from a set of keys to a set of values and supports the following methods: \textit{put}: The method call \( m.put(k, v) \) adds or updates the mapping of the key \( k \) to the value \( v \) (\( v \neq \bot \)) in the map \( m \). \textit{get}: The method call \( m.get(k) \) 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 \textit{put} and \textit{get} method calls on \( m \) where every \textit{get} method call returns \( \bot \) if there is no preceding \textit{put} method call with the same key argument; otherwise it returns the second argument of the latest preceding \textit{put} method call with the same key argument. Note that it is assumed that the map is initially empty.
Basic Map. A basic set is a basic instance of map type.
Let BasicMap denote the type of basic maps.
Let us define

\[
isXGet_{X,m}(l) = \begin{cases} 
  l \in X \land \text{obj}_X(l) = m \land \text{name}_X(l) = \text{get} 
\end{cases}
\]

\[
isXPut_{X,m}(l) = \begin{cases} 
  l \in X \land \text{obj}_X(l) = m \land \text{name}_X(l) = \text{put} 
\end{cases}
\]

\[
isXPutter_{X,m}(l_p,l_d) \iff
\begin{cases} 
  \text{isXPut}_{X,m}(l_p) \land \text{arg}_1X(l_p) = \text{arg}_1X(l_d) \land l_p <_X l_d \land \forall l'_p: \text{isXPut}_{X,m}(l'_p) \land \text{arg}_1X(l'_p) = \text{arg}_1X(l_d) \Rightarrow (l'_p \leq_X l_p \lor l_d <_X l'_p) 
\end{cases}
\]

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

\[
\forall m \in \text{BasicMap}: \forall X \in H_B(m): X \in \text{Sequential} \Rightarrow
\]

\[
\forall l_d: \text{isXGet}_{X,m}(l_d) \land \neg(\text{ret}_X(l_d) = \bot) \Rightarrow
\exists l_p: \text{isPutter}_{X,m}(l_p,l_d) \land \text{arg}_2X(l_p) = \text{ret}_X(l_d)
\]

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

\[
\forall m \in \text{BasicMap}: \forall X \in H_B(m): X \in \text{Sequential} \Rightarrow
\]

\[
\forall l_g,l_p: \text{isXGet}_{X,m}(l_g) \land \text{isPutter}_{X,m}(l_p,l_g) \land
\Rightarrow \text{ret}_X(l_g) = \text{arg}_2X(l_p)
\]
10.3 Transactional Histories

Transactional Memory. The transactional memory is a singleton object \textit{mem} that encapsulates a set of locations where each location, \(i \in I, I = \{1, \ldots, m\}\) encapsulates a value \(v\). The object \textit{mem} has five methods \textit{init}(), \textit{read}((i), \textit{write}((i, v)), \textit{commit}() and \textit{abort}(). The parameter \(t\) is the invoking transaction identifier. The method call \textit{init}() initializes \(t\) and returns \textit{ok}. The method call \textit{read}((i) returns the value of location \(i\) or aborts \(t\) and returns \textit{A}. The method \textit{write}((i, v) writes \(v\) to location \(i\) and returns \textit{ok} or aborts \(t\) and returns \textit{A}. The method \textit{commit}() tries to commit transaction \(t\). If \(t\) is successfully committed, it returns \(C\); otherwise, it returns \(A\). The method \textit{abort}() aborts \(t\) and returns \textit{A}. The object \textit{mem} can be implicit, that is \textit{mem}. \textit{read}((i) abbreviates \textit{mem}. \textit{read}((T, \textit{read}((i)) abbreviates \textit{mem}. \textit{read}((T, \textit{write}((i, v)) writes \(v\) to location \(i\) and returns \textit{ok} or aborts \(t\) and returns \textit{A}. The reserved values \textit{ok}, \textit{A}, \textit{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 \(H|\text{mem} = H_{\text{Init}} \cdot H'\) with the following conditions. \(H_{\text{Init}}\) is the following history that initializes every location to \(v_0\). \(H_{\text{Init}} = l_0 \triangleright \text{init}_0() \cdot l_{00} \triangleright \text{write}_0(1, v_0):\text{ok} \cdot \ldots \cdot l_{0m} \triangleright \text{write}_0(m, v_0):\text{ok} \cdot l_{0c} \triangleright \text{commit}_0;C\). For every \(T \in H'\), the history \(H'|T\) is a prefix of \(E.E'\). The event sequence \(E\) is the initialization method call \(l \triangleright \text{init}_T()\) (for some \(l\)), and then a sequence of reads \(l \triangleright \text{read}_T(i):v\) and writes \(l \triangleright \text{write}_T(i, v)\) (for some \(l, i, \) and \(v\)). The event sequence \(E'\) is one of the following sequences (for some \(l, i, \) and \(v\)): (1) \(\text{inv}(l \triangleright \text{read}_T(i)), \text{ret}(l \triangleright \textit{A})\), (2) \(\text{inv}(l \triangleright \text{write}_T(i, v))), \text{ret}(l \triangleright \textit{A})\), (3) \(\text{inv}(l \triangleright \text{commit}_T()), \text{ret}(l \triangleright \textit{C})\), (4) \(\text{inv}(l \triangleright \text{commit}_T()), \text{ret}(l \triangleright \textit{A})\), or (5) \(\text{inv}(l \triangleright \text{abort}_T())\), \text{ret}(l \triangleright \textit{A})\). Let \textit{THistory} denote the set of transaction histories. Let \textit{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 \(\pi\), let \(\mathbb{H}(\pi)\) denote the set of complete transaction histories that result from execution of transactions with \(\pi\).
11 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:

\[
\Gamma ::= \cdot | \Gamma ; A \quad \text{Context}
\]

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

11.1 Classical First-order Logic Inference Rules

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

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

\[
\begin{array}{ll}
\text{Premise} & \text{NegIntro} \\
\pi, \Gamma; A; \Gamma' \vdash A & \pi, \Gamma; A \vdash A' \\
\pi, \Gamma; A \vdash A & \pi, \Gamma; A \vdash \neg A' \\
\pi, \Gamma \vdash A & \pi, \Gamma \vdash \neg A \\
\end{array}
\]

\[
\begin{array}{ll}
\text{ConjIntro} & \text{ExcMid} \\
\pi, \Gamma \vdash A & \pi, \Gamma \vdash A \\
\pi, \Gamma \vdash A' & \pi, \Gamma \vdash A' \\
\pi, \Gamma \vdash A \land A' & \pi, \Gamma \vdash A \lor \neg A \\
\end{array}
\]

\[
\begin{array}{ll}
\text{ConjElimL} & \text{ConjElimR} \\
\pi, \Gamma \vdash A \land A' & \pi, \Gamma \vdash A \land A' \\
\pi, \Gamma \vdash A & \pi, \Gamma \vdash A' \\
\pi, \Gamma \vdash A & \pi, \Gamma \vdash A \\
\pi, \Gamma \vdash A \land A' & \pi, \Gamma \vdash A \lor A' \\
\end{array}
\]

\[
\begin{array}{ll}
\text{DisjIntroL} & \text{DisjIntroR} \\
\pi, \Gamma \vdash A & \pi, \Gamma \vdash A' \\
\pi, \Gamma \vdash A \lor A' & \pi, \Gamma \vdash A \lor A' \\
\end{array}
\]

\[
\begin{array}{ll}
\text{NegIntro} & \text{NegElim} \\
\pi, \Gamma; A \vdash \neg A & \pi, \Gamma \vdash A \\
\pi, \Gamma; \neg A \vdash A & \pi, \Gamma \vdash \neg A \\
\end{array}
\]

\[
\begin{array}{ll}
\text{UnivIntro} & \text{UnivElim} \\
\pi, \Gamma \vdash A[l := l] & \pi, \Gamma \vdash \forall \ell : A[l := l] \\
\pi, \Gamma \vdash A & \pi, \Gamma \vdash \forall \ell ; A \\
\end{array}
\]

\[
\begin{array}{ll}
\text{ExistIntro} & \text{ExistElim} \\
\pi, \Gamma \vdash A[l := l] & \pi, \Gamma \vdash \exists \ell ; A \\
\pi, \Gamma \vdash A & \pi, \Gamma \vdash \exists \ell ; A \\
\end{array}
\]

\[
\begin{array}{ll}
\text{CondIntro} & \text{CondElim} \\
\pi, \Gamma ; A \vdash A' & \pi, \Gamma \vdash A \\
\pi, \Gamma ; A \Rightarrow A' & \pi, \Gamma \vdash A[l := l] \vdash A' \\
\end{array}
\]

\[
\begin{array}{ll}
\text{ExistIntro} & \text{ExistElim} \\
\pi, \Gamma \vdash \exists \ell ; A & \pi, \Gamma \vdash A[l := l] \vdash A' \\
\pi, \Gamma \vdash A & \pi, \Gamma \vdash \exists \ell ; A \\
\end{array}
\]

Figure 16. Classical Inference Rules
DisjSyllL
\[ \pi, \Gamma \vdash A \lor A' \quad \pi, \Gamma \vdash \neg A \quad \pi, \Gamma \vdash A' \]

DisjSyllR
\[ \pi, \Gamma \vdash A \lor A' \quad \pi, \Gamma \vdash \neg A' \quad \pi, \Gamma \vdash A \]

CondElim'
\[ \pi, \Gamma \vdash A \Rightarrow A' \quad \pi, \Gamma \vdash \neg A' \quad \pi, \Gamma \vdash \neg A \]

Figure 17. Derived Classical Inference Rules

LRefl
\[ \pi, \Gamma \vdash l = l \]

ERS
\[ \pi, \Gamma \vdash e = e \]

LSubs
\[ \pi, \Gamma \vdash l = l' \quad \pi, \Gamma \vdash A \]
\[ \pi, \Gamma \vdash A[l := l'] \]

ERF
\[ \pi, \Gamma \vdash e = e' \]

LSubs
\[ \pi, \Gamma \vdash e = e' \quad \pi, \Gamma \vdash A \]
\[ \pi, \Gamma \vdash A[e := e'] \]

ZERO
\[ \pi, \Gamma \vdash \neg(1 = 0) \]

Figure 18. Equivalence and Arithmetic Rules

LSym
\[ \pi, \Gamma \vdash l = l' \]

LTrans
\[ \pi, \Gamma \vdash l = l' \quad \pi, \Gamma \vdash l'' \quad \pi, \Gamma \vdash e = e' \]
\[ \pi, \Gamma \vdash l'' = l' \quad \pi, \Gamma \vdash e = e' \quad \pi, \Gamma \vdash e'' \]

ESym
\[ \pi, \Gamma \vdash e = e' \]

ESym
\[ \pi, \Gamma \vdash e = e' \quad \pi, \Gamma \vdash e'' \quad \pi, \Gamma \vdash e = e'' \]

Figure 19. Derived Equivalence and Arithmetic Rules
11.2 Structure Inference Rules

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

The rule Id states that components of method calls in the history originate from components of method calls in the program. The object, arguments and other components of an executed method call labeled $\varsigma \ c$ 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 this method call annotated with $c'$. The pre-label $\varsigma$ is $\epsilon$ when $c$ is the annotation of a this method call.

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

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

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

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

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

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

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

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

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

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

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

The rule IX2OX states that if two method calls in the body of two this method calls execute in order by the same thread, then the two this method calls execute in the same order.
\(\text{Id} \quad obj_\pi(c) = \emptyset \quad \text{name}_\pi(c) = n\)
\(\text{thread}_\pi(c) = \tau \quad \text{arg}_\pi(c) = u \quad \text{retv}_\pi(c) = x\)
\(\pi, \Gamma \vdash \text{exec}(\cdot)\)
\(\pi, \Gamma \vdash \text{obj}(\cdot) = \cdot^\theta \land \text{name}(\cdot) = n \land \text{thread}(\cdot) = \cdot^\tau \land \text{arg}(\cdot) = \cdot^u \land \text{retv}(\cdot) = \cdot^x\)

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

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

\(\text{IControl}\)
\(\text{Labels}(\text{name}_\pi(c)) = \{\theta\}\)
\(\text{PreReturns}_\pi(\cdot) = \{\theta\}\)
\(\pi, \Gamma \vdash \text{exec}(\cdot c') \iff (\text{exec}(\cdot) \land \bigvee_{c_i} \cdot c_i = c_i \land \cdot c' \text{cond}_\pi(\cdot c') \land \bigwedge_{c_r} \neg \text{exec}(\cdot c_r))\)

\(\text{P2X}\)
\(c_1 \rightarrow_\pi c_2\)
\(\pi, \Gamma \vdash \text{exec}(\cdot c_1) \quad \pi, \Gamma \vdash \text{exec}(\cdot c_2)\)
\(\pi, \Gamma \vdash \cdot c'_1 < \cdot c'_2\)

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

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

\(\text{CLE}\)
\(\pi, \Gamma \vdash \tau < \tau'\)
\(\pi, \Gamma \vdash \text{exec}(l) \land \text{thread}(l) = \tau\)
\(\pi, \Gamma \vdash \text{exec}(l') \land \text{thread}(l') = \tau'\)
\(\pi, \Gamma \vdash l < l'\)

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

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

\(\text{IX2OX}\)
\(\pi, \Gamma \vdash c_1 c_3 < c_2 c_4\)
\(\pi, \Gamma \vdash \text{thread}(c_1 c_3) = \text{thread}(c_2 c_4)\)
\(\pi, \Gamma \vdash c_1 < c_2 \lor c_1 = c_2\)

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

\(\text{Figure 21.} \text{ Derived Structure Inference Rules}\)
The basic inference rules axiomatize the properties of the execution and linearization orders and their interdependence. The basic inference rules state are presented in \ref{fig:basic_inference_rules}. The derived basic inference rules are presented in Figure \ref{fig:derived_inference_rules}. We explain each rule in turn.

The rule \textsc{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 \textsc{XTrans} states the transitivity property of the precedence execution order. The rule \textsc{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 \( l_4 \).

\begin{figure}[h]
\centering
\begin{align*}
\text{XASym} & \quad \pi, \Gamma \vdash l < l' \\
\pi, \Gamma & \vdash \neg(l' < l) \land \neg(l' \prec l) \land \neg(l = l') \\
\text{XTrans} & \quad \pi, \Gamma \vdash l < l' \quad \pi, \Gamma \vdash l' < l'' \\
\pi, \Gamma & \vdash l < l'' \\
\text{XXTrans} & \quad \pi, \Gamma \vdash l_1 < l_2 \quad \pi, \Gamma \vdash l_3 < l_4 \\
\pi, \Gamma & \vdash l_2 \prec l_3 \\
\text{XTotal} & \quad \pi, \Gamma \vdash \text{exec}(l) \land \text{exec}(l') \\
\pi, \Gamma & \vdash (l < l') \lor (l' < l) \lor (l \sim l') \lor (l = l') \\
\text{X2L} & \quad \mathcal{T}_{\text{base}}(o) \in LT \\
\pi, \Gamma \vdash \text{obj}(l) = \text{obj}(l') = o \\
\pi, \Gamma & \vdash l <_o l' \\
\text{XLTrans} & \quad \mathcal{T}_{\text{base}}(o) \in LT \\
\pi, \Gamma \vdash l < l_2 \quad \pi, \Gamma \vdash l_3 < l_4 \\
\pi, \Gamma & \vdash l_2 <_o l_3 \\
\text{L2X} & \quad \mathcal{T}_{\text{base}}(o) \in LT \cup SCT \\
\pi, \Gamma \vdash l < l' \\
\pi, \Gamma & \vdash \text{exec}(l) \land \text{exec}(l') \\
\text{L2X} & \quad \mathcal{T}_{\text{base}}(o) \in LT \\
\pi, \Gamma \vdash l < l' \\
\pi, \Gamma & \vdash \text{exec}(l) \land \text{exec}(l') \land \text{obj}(l) = \text{obj}(l') = o
\end{align*}
\caption{Basic Inference Rules}
\end{figure}

\begin{figure}[h]
\centering
\begin{align*}
\pi, \Gamma \vdash c_1 \rightarrow c_2 \\
\mathcal{T}_{\text{base}}(o) \in LT \\
\pi, \Gamma \vdash \text{obj}(c_1) = \text{obj}(c_2) = o \\
\pi, \Gamma & \vdash c_1 \prec_o c_2 \\
\text{X2X} & \quad \pi, \Gamma \vdash l < l' \\
\pi, \Gamma & \vdash \text{exec}(l) \land \text{exec}(l') \\
\text{L2X} & \quad \mathcal{T}_{\text{base}}(o) \in LT \cup SCT \\
\pi, \Gamma \vdash l < l' \\
\pi, \Gamma & \vdash \text{exec}(l) \land \text{exec}(l') \land \text{obj}(l) = \text{obj}(l') = o
\end{align*}
\caption{Derived Basic Inference Rules}
\end{figure}
The rule XTotal states the totality property of the precedence and concurrency execution relations. Every two method calls either execute in order or concurrently.

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

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

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

The rule LTrans states the transitivity property of linearization orders.

The rule LTotal states the totality property of linearization orders.

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

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

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

The rule X2L’ states the contra-positive of the rule X2L.
11.4 Synchronization Object Inference Rules

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

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

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

A method call $ℓ$ is race-free isRaceFree($ℓ$) 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.

\[
\text{isRead}_r(ℓ_R) \Leftrightarrow \begin{cases} 
\text{exec}(ℓ_R) \land \text{obj}(ℓ_R) = r \land \text{name}(ℓ_R) = \text{read} \\
\text{exec}(ℓ_W) \land \text{obj}(ℓ_W) = r \land \text{name}(ℓ_W) = \text{write} 
\end{cases}
\]

\[
\text{isWrite}_r(ℓ_W, ℓ_R) \Leftrightarrow \begin{cases} 
\text{isWrite}_r(ℓ_W) \land ℓ_W < r \land ℓ_R \land \\
∀ ℓ'_W : \text{isWrite}_r(ℓ'_W) \Rightarrow (ℓ'_W ≼ r \lor ℓ_R ≻ r) \land \\
isEWrite_r(ℓ_W, ℓ_R) \Leftrightarrow \begin{cases} 
\text{isWrite}_r(ℓ_W) \land ℓ_W < ℓ_R \land \\
∀ ℓ'_W : \text{isWrite}_r(ℓ'_W) \Rightarrow (ℓ'_W ≤ ℓ_W \lor ℓ_R < ℓ'_W) 
\end{cases}
\end{cases}
\]

\[
isSequential(o) \Leftrightarrow \begin{cases} 
∀ ℓ, ℓ' : (\text{exec}(ℓ) \land \text{exec}(ℓ') \land \text{obj}(ℓ) = o \land \text{obj}(ℓ') = o) \Rightarrow \\
(ℓ ≤ ℓ' \lor ℓ < ℓ)
\end{cases}
\]

\[
isRaceFree_r(ℓ) \Leftrightarrow \begin{cases} 
∀ ℓ_W : \text{isWrite}_r(ℓ_W) \Rightarrow (ℓ_W < ℓ \lor ℓ < ℓ_W) 
\end{cases}
\]

\[
isSequentiallyWritten(r) \Leftrightarrow \begin{cases} 
∀ ℓ_w : \text{isWrite}_w(ℓ_w) \Rightarrow \text{isRaceFree}_r(ℓ_w)
\end{cases}
\]

**Figure 24.** Register Inference Rules.

\[
\text{BReg} \quad \begin{cases} 
∀ ℓ_t \exists ℓ_W : \text{isWriter}_reg(ℓ_W, ℓ_R) \land \\
\text{retv}(ℓ_R) = \text{arg1}(ℓ_W)
\end{cases}
\]

**Figure 25.** Derived Register Inference Rules.
The rule BR\textsuperscript{reg} 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 \cite{26}.

The derived register inference rules are presented in Figure 25.

The rule AR\textsubscript{reg}’ 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 \( \text{isSequential}(\textit{o}) \) if and only if every pair of method calls on it are ordered in the execution order.

The rule BR\textsubscript{reg}’ 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\textsubscript{reg} 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 cas register inference rules are presented in Figure 26.

A method call $\ell_w$ on an atomic cas register $r$ is a successful write $isWrite_r(\ell_w)$, if and only if it is a write method call or a successful cas method call. The written value $writtenValue(\ell)$ of a successful write method call $\ell$ 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 $l_R$ on an atomic cas register, there is a successful write $\ell_w$ that writes the same value that $l_R$ has returned and $\ell_w$ is the last successful write that is linearized before $l_R$.

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

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

The rule CASRegRead’ states that for every read method call $l_R$ on an atomic cas register, the last successful write that is linearized before $l_R$ writes the same value that $l_R$ returns.
Lock and Try-Lock. The preliminary definitions are presented in Figure 28 and the lock and try-lock inference rules are presented in Figure 29.

Ownership for a lock \( l \) is respected, \( \text{isOwnerRespecting}(l) \) 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 \( f a l s e \), 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 \( f a l s e \), 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 \( t r u e \).

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.

\[
\text{isLock}_o(\ell) \iff \text{exec}(\ell) \land \text{obj}(\ell) = o \land \text{name}(\ell) = \text{lock}
\]
\[
\text{isUnlock}_o(\ell) \iff \text{exec}(\ell) \land \text{obj}(\ell) = o \land \text{name}(\ell) = \text{unlock}
\]
\[
\text{isRead}_o(\ell) \iff \text{exec}(\ell) \land \text{obj}(\ell) = o \land \text{name}(\ell) = \text{read}
\]
\[
\text{isTryLock}_o(\ell) \iff \text{exec}(\ell) \land \text{obj}(\ell) = o \land \text{name}(\ell) = \text{tryLock}
\]
\[
\text{isTLock}_o(\ell) \iff \text{isLock}_o(\ell) \lor \text{isTryLock}_o(\ell) \land \text{retv}(\ell) = \text{true}
\]
\[
\text{noUnlockBetween}_o(\ell_1, \ell_u) \iff \forall \ell': \text{isUnlock}_o(\ell') \Rightarrow \exists \ell': \text{isTLock}_o(\ell') \land \text{thread}(\ell') = \text{thread}(\ell) \land \ell' < \ell \land
\]
\[
\forall \ell'': (\text{isUnlock}_o(\ell'') \land \text{thread}(\ell'') = \text{thread}(\ell)) \Rightarrow \ell'' < \ell' \lor \ell \leq \ell''
\]

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

\[
\begin{align*}
\text{SCounter} & \quad \mathcal{T}_{base}(o) = \text{SCounter} \\
& \quad \pi, \Gamma \vdash \text{obj}(l_1) = o \\
& \quad \pi, \Gamma \vdash \text{obj}(l_2) = o \land \text{name}(l_2) = \text{iaf} \\
& \quad \pi, \Gamma \vdash l_1 \preceq_o l_2 \\
& \quad \pi, \Gamma \vdash \text{retv}(l_1) < \text{retv}(l_2)
\end{align*}
\]

**Figure 30.** SCounter Rules

\[
\begin{align*}
\text{SCounter’} & \quad \mathcal{T}_{base}(o) = \text{SCounter} \\
& \quad \pi, \Gamma \vdash \text{exec}(l_1) \land \text{obj}(l_1) = o \\
& \quad \pi, \Gamma \vdash \text{exec}(l_2) \land \text{obj}(l_2) = o \land \text{name}(l_2) = \text{iaf} \\
& \quad \pi, \Gamma \vdash \text{retv}(l_1) > \text{retv}(l_2) \\
& \quad \pi, \Gamma \vdash l_2 \preceq_o l_1
\end{align*}
\]

**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 \( \text{BasicSetContains} \) 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 \( \text{BasicSetAdd} \) 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 \( \text{BasicMapGet} \) 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.

\[
\begin{align*}
\text{BasicSetContains} & \quad \mathcal{T}_{\text{base}}(s) = \text{BasicSet} \\
& \quad \pi, \Gamma \vdash \text{isSequential}(s) \\
& \quad \pi, \Gamma \vdash \exists l_a \colon \text{isAdd}_{s}(l_a) \land \arg 1(l_a) = \arg 1(l_c) \land \ell_a < \ell_c \\
& \quad \pi, \Gamma \vdash \text{retv}(l_c) = \text{true} \\
\end{align*}
\]

\[
\begin{align*}
\text{BasicSetAdd} & \quad \mathcal{T}_{\text{base}}(s) = \text{BasicSet} \\
& \quad \pi, \Gamma \vdash \text{isSequential}(s) \\
& \quad \pi, \Gamma \vdash \exists \ell_a \colon \text{isAdd}_{s}(\ell_a) \\
& \quad \pi, \Gamma \vdash \exists l_a \colon \text{isContains}_{s}(l_c) \\
& \quad \pi, \Gamma \vdash l_a < \ell_c \land \arg 1(l_a) = \arg 1(l_c) \\
& \quad \pi, \Gamma \vdash \text{retv}(l_c) = \text{true} \\
\end{align*}
\]

\[
\begin{align*}
\text{BasicMapGet} & \quad \mathcal{T}_{\text{base}}(m) = \text{BasicMap} \\
& \quad \pi, \Gamma \vdash \text{isSequential}(m) \\
& \quad \pi, \Gamma \vdash \text{isGet}_{m}(l_g) \land \text{retv}(l_g) \neq \bot \\
& \quad \pi, \Gamma \vdash \exists l_p \colon \text{isPutter}_{m}(l_p, l_g) \land \arg 2(l_p) = \text{retv}(l_g) \\
\end{align*}
\]

\[
\begin{align*}
\text{BasicMapPut} & \quad \mathcal{T}_{\text{base}}(m) = \text{BasicMap} \\
& \quad \pi, \Gamma \vdash \text{isSequential}(m) \\
& \quad \pi, \Gamma \vdash \exists l_g \colon \text{isPut}_{m}(l_g) \\
& \quad \pi, \Gamma \vdash \exists l_p \colon \text{isPutter}_{m}(l_p, l_g) \\
& \quad \pi, \Gamma \vdash \arg 2(l_p) = \text{retv}(l_g) \\
\end{align*}
\]

\[
\begin{align*}
\text{BasicMapGet}^\ast & \quad \mathcal{T}_{\text{base}}(m) = \text{BasicMap} \\
& \quad \pi, \Gamma \vdash \text{isSequential}(m) \\
& \quad \pi, \Gamma \vdash \neg \exists l_p \colon \text{isPut}_{m}(l_p) \land \arg 1(l_p) = \arg 1(l_g) \land \ell_p < \ell_g \\
& \quad \pi, \Gamma \vdash \text{retv}(l_g) = \bot \\
\end{align*}
\]

\[
\begin{align*}
\text{BasicMapPut}^\ast & \quad \mathcal{T}_{\text{base}}(m) = \text{BasicMap} \\
& \quad \pi, \Gamma \vdash \text{isSequential}(m) \\
& \quad \pi, \Gamma \vdash \exists l_g \colon \text{isPut}_{m}(l_g) \\
& \quad \pi, \Gamma \vdash \exists l_p \colon \text{isPut}_{m}(l_p) \\
& \quad \pi, \Gamma \vdash \arg 1(l_p) = \arg 1(l_g) \land l_p < l_g \\
& \quad \pi, \Gamma \vdash \forall l_p \colon \text{isPut}_{m}(l_p) \Rightarrow \text{arg2}(l_p) \neq \bot \\
& \quad \pi, \Gamma \vdash \text{retv}(l_g) \neq \bot \\
\end{align*}
\]

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

**Figure 33.** Derived Set and Map Inference Rules
The rule BasicMapPut states that if a basic map $m$ is accessed sequentially, for every get method call $l_g$ on $m$, if $l_p$ is the latest preceding put method call on $m$ with the same key argument then the value argument of $l_p$ is equal to the return value of $l_g$.

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

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

The rule BasicMapPut’ states that if a basic map $m$ is accessed sequentially and no put method call puts $\bot$ in $m$, every get method call that succeeds a put method call with the same key argument does not return $\bot$. 
Theorem 12.1 (Mutual Exclusion)

In every execution of the Dekker specification, at most one thread acquires the lock.

Proof:

We show that

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

We show that

(2) \( \text{retv}_X(L_2) = \text{true} \) \Rightarrow \( \text{retv}_X(L_1) = \text{false} \)

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

(3) \( X = (X, \sigma, L) \in \llbracket \pi \rrbracket \)
(4) \( X' = \sigma(X) \)

By Lemma 12.2, we have

(5) \( \pi_{\text{Dekker}}, \cdot \vdash \text{retv}(L_2) = \text{true} \) \Rightarrow \( \text{retv}(L_1) = \text{false} \).

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

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

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

(7) \( \text{retv}_X(L_2) = \text{true} \) \Rightarrow \( \text{retv}_X(L_1) = \text{false} \). \( \Box \)

Lemma 12.2.

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

Proof:

Let

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

We show that

\( \pi, \cdot \vdash \text{retv}(L_2) = \text{true} \) \Rightarrow \( \text{retv}(L_1) = \text{false} \)

Let

(8) \( \pi = \text{retv}(L_2) = \text{true} \)

By rule CIN, we have to show that

(9) \( \pi, \pi \vdash \text{retv}(L_2) = \text{false} \)

By rule Pre on [8], we have

(10) \( \pi, \pi \vdash \text{retv}(L_2) = \text{false} \)

From \( \pi \), we have

(11) \( \pi, \pi \vdash \text{cond}_x(L_2) = \text{true} \)

By rule OCON on [10], we have

(12) \( \pi, \pi \vdash \text{exec}(L_2) \)

From \( \pi \), we have

(13) \( \pi, \pi \vdash \text{name}_x(L_2) = \text{tryLock2} \)

(14) \( \pi, \pi \vdash \text{Labels}(\text{tryLock2}) \)

From \( \pi \), we have

(15) \( \pi, \pi \vdash \text{cond}_x(R_1) = \text{true} \)

Thus,

(16) \( \pi, \pi \vdash \text{Labels}(\text{tryLock2}) \)

From \( \pi \), we have

(17) \( \pi, \pi \vdash \text{obj}(L_2, R_1) = \text{f}_1 \)

(18) \( \pi, \pi \vdash \text{name}(L_2, R_1) = \text{read} \)

(19) \( \pi, \pi \vdash \text{arg}(L_2, R_2) = \text{1} \)

Similarly, we have

(20) \( \pi, \pi \vdash \text{exec}(L_2, W_2) \)

(21) \( \pi, \pi \vdash \text{obj}(L_2, W_2) = \text{f}_2 \)

(22) \( \pi, \pi \vdash \text{name}(L_2, W_2) = \text{write} \)

(23) \( \pi, \pi \vdash \text{arg}(L_2, W_2) = \text{1} \)

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

(24) \( \pi, \pi \vdash \text{isRead}_x(L_2, R_1) \)

From rule AREG on [24], we have

(25) \( \pi, \pi \vdash \exists C_w : \text{isWriter}_f (L_w, L_2, R_1) \land \text{arg}(L_w) = \text{retv}(L_2, R_1) \)

Let

(26) \( \pi' = \pi \)

\( \text{isWriter}_f (L_w, L_2, R_1) \land \text{arg}(L_w) = \text{retv}(L_2, R_1) \)

where \( L_w \) is fresh.

By rule Pre on [26], we have

(27) \( \pi, \pi' \vdash \text{isWriter}_f (L_w, L_2, R_1) \)

(28) \( \pi, \pi' \vdash \text{arg}(L_w) = \text{retv}(L_2, R_1) \)

By rule Pre on [11], we have

(29) \( \pi, \pi \vdash \text{obj}(L_2) = \text{true} \)

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

From \( \pi \), we have

(31) \( \text{Labels}_x(\text{tryLock2}) = \{ C_{21}, C_{2f} \} \)

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

(32) \( \pi, \pi \vdash \text{exec}(L_2, C_{21}) \land \text{arg}(L_2, C_{2f}) = \text{retv}(L_2) \lor \text{exec}(L_2, C_{2f}) \land \text{arg}(L_2, C_{2f}) = \text{retv}(L_2) \)

We apply rule DSE to [32]:

Right:

Let

(33) \( \pi' = \pi \)

\( \text{exec}(L_2, C_{2f}) \land \text{arg}(L_2, C_{2f}) = \text{retv}(L_2) \)

By rule Pre on [33], we have

(34) \( \pi, \pi \vdash \text{exec}(L_2, C_{2f}) \)
\[(35) \, \pi, \Gamma' \vdash arg1(L_2' C_{q_f}) = \text{retv}(L_2)\]
From \(\pi\), we have
\[(36) \, arg1(C_{q_f}) = \text{false}\]
By rule Id on \([34],[36]\), we have
\[(37) \, \pi, \Gamma' \vdash arg1(L_2' C_{q_f}) = \text{false}\]
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) \(\Gamma' = \Gamma\);
\[(exec(L_2' C_{q_f}) \land arg1(L_2' C_{q_f}) = \text{retv}(L_2))\]
By rule Premise on \([41]\), we have
\[(42) \, \pi, \Gamma' \vdash exec(L_2' C_{q_f})\]
\[(43) \, \pi, \Gamma' \vdash arg1(L_2' C_{q_f}) = \text{retv}(L_2)\]
From \(\pi\), we have
\[(44) \, cond_{\text{ar}}(C_{q_f}) \equiv (x_1 = 0)\]
By rule IControl on \([42]\) and \([44]\) we have
\[(45) \, \pi, \Gamma' \vdash (L_2' x_1 = 0)\]
From \([28],[19],[45]\), weakening (Lemma 13.2) and rule ETrans, we have
\[(46) \, \pi, \Gamma' \vdash arg1(l_w) = 0\]
From the definition of isWriter on \([27]\) and rule ConjELIM and rule ConjELIMR, we have
\[(47) \, \pi, \Gamma' \vdash obj(l_w) = f_1\]
\[(48) \, \pi, \Gamma' \vdash name(l_w) = \text{write}\]
\[(49) \, \pi, \Gamma' \vdash exec(l_w)\]
\[(50) \, \pi, \Gamma' \vdash l_w \ll f_1 L_2' R_1\]
\[(51) \, \pi, \Gamma' \vdash \forall \ell_{w'}: \text{isWriter}_{f_1}(\ell_{w'}) \Rightarrow \ell_{w'} \approx f_1 \ell_{w'} \lor L_2' R_1 \ll f_1 \ell_{w'}\]
From the definition of \(\pi\), we have
\[(52) \, \text{calls}_{\pi}(f_1, \text{write}) \equiv \{W_1, W_0\}\]
From rule Scc on \([47],[48],[49]\) and \([52]\), we have that for some fresh \(\zeta\)
\[(53) \, \pi, \Gamma' \vdash l_w = \zeta' W_1 \lor l_w = \zeta' W_0\]
We apply rule DisJELIM to \([53]\):
\[(54) \, \Gamma'' = \Gamma';\]
\[(55) \, l_w = \zeta' W_1\]
From \([49],[54]\), weakening (Lemma 13.2), we have
\[(55) \, \pi, \Gamma'' \vdash \text{exec}(\zeta' W_1)\]
From \(\pi\), we have
\[(56) \, arg1_\zeta(W_1) = 1\]
By rule Id on \([54],[56]\), we have
\[(57) \, \pi, \Gamma'' \vdash arg1(\zeta' W_1) = 1\]
From \([54],[57]\), we have
\[(58) \, \pi, \Gamma'' \vdash arg1(l_w) = 1\]
By weakening (Lemma 13.2) on \([46]\), we have
\[(59) \, \pi, \Gamma'' \vdash arg1(l_w) = 0\]
By rule ETrans and rule ESym on \([58],[59]\), we have
\[(60) \, \pi, \Gamma'' \vdash 0 = 1\]
By rule NegElim on rule Zero and \([60]\), we have
\[(61) \, \pi, \Gamma'' \vdash \text{retv}(L_1) = \text{false}\]
Right:
\[(62) \, \Gamma'' = \Gamma';\]
\[(63) \, l_w = \zeta' W_0\]
By rule Premise on \([62]\), we have
\[(64) \, \pi, \Gamma'' \vdash l_w = \zeta' W_0\]
From \(\pi\), we have
\[(65) \, \text{call}_{\pi}(\text{this}, \text{init}) \equiv \{L_0\}\]
By rule CallEE on \([63]\) and \([64]\) we have
\[(66) \, \pi, \Gamma'' \vdash \text{exec}(\zeta)\]
\[(67) \, \pi, \Gamma'' \vdash \text{obj}(\zeta) = \text{this}\]
\[(68) \, \pi, \Gamma'' \vdash \text{name}(\zeta) = \text{init}\]
From \(\pi\), we have
\[(69) \, \text{call}_{\pi}(\text{this}, \text{init}) = \{L_0\}\]
By rule Scc on \([65]-[69]\) we have
\[(70) \, \pi, \Gamma'' \vdash \zeta = L_0\]
From \([63]\) and \([70]\), we have
\[(71) \, \pi, \Gamma'' \vdash l_w = L_0' W_0\]
From \(\pi\), we have
\[(72) \, \pi, \Gamma'' \vdash \text{cond}_{\pi}(L_1) = \text{true}\]
Thus,
\[(73) \, \pi, \Gamma'' \vdash \text{exec}(L_1)\]
From \(\pi\), we have
\[(74) \, \text{name}_{\pi}(L_1) = \text{tryLock1}\]
\[(75) \, R_2 \in \text{Labels}\{\text{tryLock1}\}\]
From \(\pi\), we have
\[(76) \, \pi, \Gamma'' \vdash L_1' \text{cond}_{\pi}(R_2) = \text{true}\]
From \(\pi\), we have
\[(77) \, \text{PreReturns}_{\pi}(R_2) = \emptyset\]
By rule IControl on \([73]-[77]\), we have
\[(78) \, \pi, \Gamma'' \vdash \text{exec}(L_1' R_2)\]
From \(\pi\) we have
\[(79) \, \text{obj}_{\pi}(R_2) = f_2\]
\[(80) \, \text{name}_{\pi}(R_2) = \text{read}\]
\[(81) \, \text{retv}_{\pi}(R_2) = \chi_2\]
By rule Id on \([78]\) and \([79]-[81]\), and then rule ConjELIM and rule ConjELIMR, we have
\[(82) \, \pi, \Gamma'' \vdash \text{obj}(L_1' R_2) = f_2\]
\[(83) \, \pi, \Gamma'' \vdash \text{name}(L_1' R_2) = \text{read}\]
\[(84) \, \pi, \Gamma'' \vdash \text{retv}(L_1' R_2) = L_1' \chi_2\]
From the definition of isRead on \([78],[82],[83]\) and rule Con- jIntro, we have
\[(85) \, \pi, \Gamma'' \vdash \text{isRead}_{f_1}(L_1' R_2)\]
Similarly, we have that

\[(86) \quad \pi, \Gamma'' \vdash \text{exec}(L_1 W_1)\]

\[(87) \quad \pi, \Gamma'' \vdash \text{obj}(L_1 W_1) = f_i\]

\[(88) \quad \pi, \Gamma'' \vdash \text{name}(L_1 W_1) = \text{write}\]

\[(89) \quad \pi, \Gamma'' \vdash \text{arg1}(L_1 W_1) = 1\]

\[(90) \quad \pi, \Gamma'' \vdash \text{isWrite}_{f_i}(L_1 W_1)\]

By rule UnivrElim on [51], and [90], we have

\[(91) \quad \pi, \Gamma'' \vdash L_1 W_1 \preceq f_i L_0 W_0 \lor L_2 R_1 \preceq f_i L_1 W_1\]

By rule LSubs on [91] and [71], we have

\[(92) \quad \pi, \Gamma'' \vdash L_1 W_1 \preceq f_i L_0 W_0 \lor L_2 R_1 \preceq f_i L_1 W_1\]

From \(\pi\), we have

\[(93) \quad L_0 \to_{\pi} L_1\]

By rule LSubs on [66] and [70], we have

\[(94) \quad \pi, \Gamma'' \vdash \text{exec}(L_0)\]

By rule P2X on [93], [94] and [73], we have

\[(95) \quad \pi, \Gamma'' \vdash L_0 \prec f_i L_1 W_1\]

By rule LSubs on [49] and [71], we have

\[(96) \quad \pi, \Gamma'' \vdash \text{exec}(L_0 W_0)\]

By rule OX2IX on [95], [96], and [86], we have

\[(97) \quad \pi, \Gamma'' \vdash L_0 W_0 \prec f_i L_1 W_1\]

By rule Ldo on [96], we have

\[(98) \quad \pi, \Gamma'' \vdash \text{obj}(L_0 W_0) = f_i\]

By rule X2L on [97], [98] and [87], we have

\[(99) \quad \pi, \Gamma'' \vdash L_0 W_0 \prec f_i L_1 W_1\]

By rule LASym on [99], and rule ConjrElml, we have

\[(100) \quad \pi, \Gamma'' \vdash \neg((L_1 W_1 \prec f_i L_0 W_0))\]

By rule DjsjEll on [92], [100], we have

\[(101) \quad \pi, \Gamma'' \vdash L_2 R_1 \prec f_i L_1 W_1\]

From \(\pi\), we have

\[(102) \quad W_2 \to_{\pi} R_1\]

From rule P2X on [102], [20], [16], and weakening (Lemma 13.2), we have

\[(103) \quad \pi, \Gamma'' \vdash L_2 W_2 \prec f_i L_1 R_2\]

From \(\pi\), we have

\[(104) \quad W_1 \to_{\pi} R_2\]

From rule P2X on [104], [86] and [78], we have

\[(105) \quad \pi, \Gamma'' \vdash L_1 W_1 \prec f_i L_1 R_2\]

From rule XlTrans on [103], [101] and [105], we have

\[(106) \quad \pi, \Gamma'' \vdash L_2 W_2 \prec f_i L_1 R_2\]

From rule X2L on [106], [21] and [82], we have

\[(107) \quad \pi, \Gamma'' \vdash L_2 W_2 \prec f_i L_1 R_2\]

We show that

\[(108) \quad \pi, \Gamma'' \vdash \forall \ell_W:\]

\[\text{isWrite}_{f_i}(\ell_W) \Rightarrow \]

\[\ell_W \preceq f_i L_2 W_2 \lor L_1 R_2 \preceq f_i \ell_W\]

Let

\[(109) \quad \Gamma''' = \Gamma'''; \text{isWrite}_{f_i}(\ell_W)^\prime\]

By rule UnivrIntro and rule ConjrIntro, we have to show that

\[\pi, \Gamma''' \vdash \ell_W \preceq f_i L_2 W_2 \lor L_1 R_2 \preceq f_i \ell_W\]

By rule Premise on [109], we have

\[(110) \quad \pi, \Gamma''' \vdash \text{isWrite}_{f_i}(\ell_W)^\prime\]

From definition of isWrite on [110], we have

\[(111) \quad \pi, \Gamma''' \vdash \text{obj}(\ell_W) = f_i \land \]

\[\text{name}(\ell_W) = \text{write} \land \]

\[\text{exec}(\ell_W)^\prime\]

From the definition of \(\pi\), we have

\[(112) \quad \text{calls}_x(f_2, \text{write}) = \{W_0, W_2\}\]

By rule Src on [111] and [112], we have that for some fresh \(\varsigma\),

\[(113) \quad \pi, \Gamma''' \vdash \ell_W = \varsigma W_0 \lor \ell_W = \varsigma W_2\]

We apply rule DjsjElml on [113]:

Left:

\[(114) \quad \Gamma''' = \Gamma'''; \ell_W = \varsigma W_0\]

By rule Premise on [114], we have

\[(115) \quad \pi, \Gamma''' \vdash \ell_W = \varsigma W_0\]

By rule LSubs on [111], [115] and weakening (Lemma 13.2), we have

\[(116) \quad \pi, \Gamma''' \vdash \text{exec}(\varsigma W_0)\]

From \(\pi\), we have

\[(117) \quad W_0 \in \text{Labels}_x(\text{init})\]

By rule Callee on [116] and [117], we have

\[(118) \quad \pi, \Gamma''' \vdash \neg(\varsigma = e)\]

\[(119) \quad \pi, \Gamma''' \vdash \text{exec}(\varsigma)\]

\[(120) \quad \pi, \Gamma''' \vdash \text{obj}(\varsigma) = \text{this}\]

\[(121) \quad \pi, \Gamma''' \vdash \text{name}(\varsigma) = \text{init}\]

From \(\pi\), we have

\[(122) \quad \text{calls}_x(\text{this}, \text{init}) = \{L_0\}\]

By rule Src on [118]-[122], we have

\[(123) \quad \pi, \Gamma''' \vdash \varsigma = L_0\]

By rule LSubs on [115], [123], we have

\[(124) \quad \pi, \Gamma''' \vdash \ell_W = L_0 W_0\]

By rule LSubs on [111], [124], we have

\[(125) \quad \pi, \Gamma''' \vdash \text{obj}(L_0 W_0) = f_i\]

\[(126) \quad \pi, \Gamma''' \vdash \text{exec}(L_0 W_0)\]

From \(\pi\), we have

\[(127) \quad L_0 \to_{\pi} L_2\]

By rule P2X on [127], [94] and [11], weakening (Lemma 13.2), we have

\[(128) \quad \pi, \Gamma''' \vdash L_0 < L_2\]

By rule OX2IX on [128], and [126], and [20], we have

\[(129) \quad \pi, \Gamma''' \vdash L_0 W_0 < L_2 W_2\]

By rule X2L on [129], and [125], and [21], we have

\[(130) \quad \pi, \Gamma''' \vdash L_0 W_0 < f_i L_2 W_2\]

By rule DjsjIntroD on [130], we have

\[(131) \quad \pi, \Gamma''' \vdash L_0 W_0 < f_i L_2 W_2 \lor L_1 R_2 \preceq f_i L_0 W_0\]

By rule LSubs on [131] and [124], we have

\[(132) \quad \pi, \Gamma''' \vdash \]
\[ \Gamma'''' \vdash \Gamma'''' \vdash \zeta'''' \vdash \zeta'''' \vdash \zeta = L_2 \]

By rule PREMISE on \([133]\), we have
\[ \pi, \Gamma'''' \vdash I''''_W = \zeta''''_W \]

Similar to the previous part, we can show that
\[ \pi, \Gamma'''' \vdash \varsigma = L_2 \]

By rule LSUBS on \([134]\) and \([135]\), we have
\[ \pi, \Gamma'''' \vdash I''''_W = L_2''''_W \]

By rule DISJINTRO on \([136]\), we have
\[ \pi, \Gamma'''' \vdash I''''_W \leq \Gamma, L_2''''_W \]

Thus, by rule DISJINTROL on \([137]\), we have
\[ \pi, \Gamma'''' \vdash I''''_W \leq \Gamma, L_2''''_W \]

By rule CONJINTRO and the definition of isWriter on \([20]\)-\([22]\) and weakening (Lemma 13.2), we have
\[ \pi, \Gamma'''' \vdash \text{isWriter}_{\Gamma, L_2''''_W} \]

By rule CONJINTRO and the definition of isWriter on \([138]\), \([107]\), and \([108]\), we have
\[ \pi, \Gamma'''' \vdash \text{isWriter}_{\Gamma, L_2''''_W, L_1''''_R_2} \]

By rule CONJINTRO and the definition of isRead on \([78]\), \([82]\) and \([83]\), we have
\[ \pi, \Gamma'''' \vdash \text{isRead}_{\Gamma, L_1''''_R_2} \]

From rule AReg on \([140]\) and \([139]\), we have
\[ \pi, \Gamma'''' \vdash \text{retv}(L_1''''_R_2) = \text{arg}1(L_2''''_W) \]

By rule ETRANS and rule ESYM on \([141]\), \([84]\) and \([23]\), we have
\[ \pi, \Gamma'''' \vdash \text{L'}_{1''''} = 1 \]

By rule ZERO and rule ESYM on \([142]\), we have
\[ \pi, \Gamma'''' \vdash \neg(L_1'''' \text{retv}) \]

From \(\pi\), we have that
\[ \pi, \Gamma'''' \vdash \neg(x_2 = 0) \]

From \([143]\) and \([148]\), we have
\[ \pi, \Gamma'''' \vdash \text{Cond}_{\Gamma, (C_y)} = \emptyset \]

From \([144]\), we have
\[ \pi, \Gamma'''' \vdash \text{Label}_{\Gamma, (C_y)}(\text{tryLock}1) \]

From \([145]\), \([146]\), \([147]\), \([149]\), \([150]\), \([151]\), \([152]\), \([153]\), \([154]\), \([155]\), \([156]\) and \([157]\), we have
\[ \pi, \Gamma'''' \text{Ret} \]

From \(\pi\), we have that
\[ \pi, \Gamma'''' \text{Con} \]

\(\square\)
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).
\[ \forall \pi, \Gamma, \Gamma', A, A', A'': (\pi, \Gamma; A; A'; \Gamma' \vdash A'') \Rightarrow (\pi, \Gamma; A'; A; \Gamma' \vdash A'') \]

Lemma 13.2 (Weakening).
\[ \forall \pi, \Gamma, A, A': (\pi, \Gamma \vdash A) \Rightarrow (\pi, \Gamma; A' \vdash A) \]

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

Definition 13.3. A specification \( \pi \) models an assertion \( A \) if and only if every execution of \( \pi \) models \( A \).
\[ \pi \models A \iff \forall X \in [[\pi]] : X \models A \]

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

Theorem 13.4 (Soundness).
\[ \forall \pi, A : ((\pi, \Gamma \vdash A) \land (\pi \models \Gamma)) \Rightarrow (\pi \models A) \]

See Section 15.2 for the proof.
14 Client Assertions

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

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

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

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

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

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

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

\[ \Gamma_7 = \forall t: \text{isCommitted}(t) \lor \text{isAborted}(t) \quad (143) \]

**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) \( rEv(l) \prec_X iEv(l') \)

As \( X \) is a valid history, we have
(4) \( iEv(l) \prec_X rEv(l) \)
(5) \( iEv(l') \prec_X rEv(l') \)

From [4], [3], and [5], we have
(6) \( iEv(l) \prec_X rEv(l') \)

From [6], we have
(7) \( \neg (rEv(l') \prec_X iEv(l)) \)

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) \( rEv(l_1) \prec_X iEv(l_2) \)
From [2], we have
(5) \( rEv(l_3) \prec_X iEv(l_4) \)
From [3], we have
(6) \( \neg (l_3 \prec_X l_2) \)
From [6], we have
(7) \( \neg (rEv(l_3) \prec_X iEv(l_2)) \)
From [7], we have
(8) \( iEv(l_2) \prec_X rEv(l_3) \)
From [4], [8], and [5], we have
(9) \( rEv(l_3) \prec_X iEv(l_4) \)
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 $\triangleleft_X$ and $\triangleleft_X$.

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

### 15.1.2 Synchronization Object Types

Lemma 10.11: 
Straightforward from $\triangleleft_X \subseteq \triangleleft_L$.

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

Lemma 10.13: 
We have
1. $l <_L l'$
2. $rEv(l) <_L iEv(l')$
From the well-formedness of the history $O$, we have
3. $iEv(l) <_L rEv(l)$
4. $iEv(l') <_L rEv(l')$
From [3], [2] and [4], we have
5. $iEv(l) <_L rEv(l')$
From [5], we have
6. $\neg(rEv(l') <_L iEv(l))$
From [2] and [6], we have
7. $\neg(l' = l)$
From the definition of $\triangleleft_X$ on [6], we have
8. $\neg(l' <_L l)$
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
$l <_L l' \lor l' <_L l \lor l = l'$
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. \( l <_L l' \)

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
\( l \in X \)
\( l' \in X \)

From [1] on [6] and [7], we have
\( \text{obj}_X(l) = o \)
\( \text{obj}_X(l') = o \)

Lemma 10.17:
Using \( L2X \) and \( X\text{Total} \), we have four cases:
Case: \( l \prec l' \)
Straightforward from \( X\text{Trans} \).
Case: \( l \sim l' \)
Straightforward from \( XX\text{Trans} \).
Case: \( l' \prec l \)
Straightforward from \( X\text{2L} \) and \( \text{LASym} \).
Case: \( l' = l \)
Straightforward from \( \text{LASym} \).

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 = p_0, p_1 ... p_n
(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
X' ∈ H(π)

By definition [13] on [6], we have
(9) ∀o: T_{base}(o) ∈ BT ⇒
X'|o ∈ H_B(o)
(10) ∀o: T_{base}(o) ∈ LT ⇒
(X'|o, L(o)) ∈ H_L(o)

∃X_1, ..., X_n:
(11) ∀i ∈ [0..n]: (X_i, σ) ∈ [[p_i]] ∧
(12) X'' ∈ Interleave(X_1, ..., X_n) ∧
X = X_0 · X''

By definition [85] on [9], we have
(13) ∀o: o ∈ T_{base}(o) ∈ BT ⇒
(X'|o ∈ Sequential) ⇒
(X'|o ∈ SeqSpec(o))

By definition [87] on [10], we have
(14) ∀o: o ∈ T_{base}(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
(15) T_{base}(o) ∈ LT
(16) π, Γ ⊢ l < l'
(17) π, Γ ⊢ obj(l) = obj(l') = o
(18) 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) obj_X(l) = obj_X(l') = σ(o)
From [19] and [20], we have
(21) l <_{X|σ(o)} l'

By [15], we have
(22) T_{base}(σ(o)) ∈ LT
By [10] and [22], we have
(23) (X'|σ(o), L(σ(o))) ∈ H_L(o)
By Lemma 10.11 on [23] and [21], we have
l <_{L(σ(o))} l'

Case rule Src:
We have that
(24) A = \bigvee_{i=1..n} c = c_i
(25) π, Γ ⊢ exec(\zeta(c'))
(26) π, Γ ⊢ obj(\zeta(c')) = 0
(27) π, Γ ⊢ name(\zeta(c')) = n
(28) Calls_{\zeta}(basename(\theta), n) = \{c_i\}

We show that
A ⊨ \bigvee_{i=1..n} c = c_i
that is
\bigvee_{i=1..n} c = c_i

By the induction hypothesis on [25], [26],
[27], and then [2], [6] and [7], we have
(29) \zeta' c ∈ X'
(30) obj_X(\zeta' c) = \zeta' \theta
(31) name_X(\zeta' c) = n
From [7] and [12] on [29], [30], [31], we have
∃i ∈ 0..n:
(32) \zeta' c ∈ X_i
(33) obj_{X_i}(\zeta' c) = \zeta' \theta
(34) name_{X_i}(\zeta' c) = n

By Lemma 15.2 on [11] and [32], we have
(35) basename(obj_{X_i}(\zeta' c)) = obj_\zeta(c)
(36) name_{X_i}(\zeta' c) = name_\zeta(c)
By the definition of basename and \', we have
(37) basename(\zeta' \theta) = basename(\theta)
From [35], [30] and [37], we have
(38) basename(obj_\zeta(c)) = basename(\theta)
From [36] and [34], we have
(39) name_\zeta(c) = n
From the definition of calls_\zeta(basename(\theta), n)
on [38] and [39], we have
\( c \in calls_s(basename(\theta), n) \)
From [28] and [36], we have
\( \bigvee_{i=1..n} c_i = c_i \)

Case rule P2X:

We have that
\( A = \gamma'c_1 \leq \gamma'c_2 \)
\( c_1 \rightarrow_\pi c_2 \)
\( \pi, \Gamma \vdash \text{exec}(\gamma'c_1) \)
\( \pi, \Gamma \vdash \text{exec}(\gamma'c_2) \)
We show that
\( X \models \gamma'c_1 < X' \gamma'c_2 \)
that is
\( \gamma'c_1 < X' \gamma'c_2 \)

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

Case rule OX2IX:

We have that
\( A = c_1'c_3 < c_2'c_4 \)
\( \pi, \Gamma \vdash c_1 < c_2 \)
\( \pi, \Gamma \vdash \text{exec}(c_1'c_3) \)
\( \pi, \Gamma \vdash \text{exec}(c_2'c_4) \)
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
\( X \models c_1 < c_2 \)
\( X \models \text{exec}(c_1'c_3) \)
\( X \models \text{exec}(c_2'c_4) \)
that is
\( c_1 < X' c_2 \)
\( c_1'c_3 \in X' \)
\( c_2'c_4 \in X' \)
From [56], we have
\( rEv(c_1) < X' iEv(c_2) \)
From Lemma 15.7 on [8] and [57], we have
\( rEv(c_1'c_3) < X' rEv(c_1) \)
From Lemma 15.7 on [8], and [58], we have
\( iEv(c_2) < X' iEv(c_2'c_4) \)

Case rule ICONTROL:

We have that
\( A = \)
\( \text{exec}(c'c') \leftrightarrow \text{exec}(c) \land \)
\( \bigvee_{i} c_i = c_i \land \)
\( \sigma(c'\text{cond}_{\pi}(c')) \land \)
\( \bigwedge_{i=1..n} \neg(c'^i_i \in X') \)
We first show that
\( c'c' \in X' \Rightarrow \)
\( c \in X' \land \)
\( \bigvee_{i} c_i = c_i \land \)
\( \sigma(c'\text{cond}_{\pi}(c')) \land \)
\( \bigwedge_{i=1..n} \neg(c'^i_i \in X') \)
We assume that
\( c'c' \in X' \)
We show that
\( c \in X' \land \)
\( \bigvee_{i} c_i = c_i \land \)
\( \sigma(c'\text{cond}_{\pi}(c')) \land \)
\( \bigwedge_{i=1..n} \neg(c'^i_i \in X') \)
From [7] and [12] on [67], we have
\( \exists i \in \{0..n\} : \)
\( c'^i_i \in X_i \)
By Lemma 15.3 on [65], [66], [11] and [68], we have
\( c \in X_i \land \)
\( \bigvee_{i} c_i = c_i \land \)
\( \sigma(c'\text{cond}_{\pi}(c')) \land \)
\( \bigwedge_{i=1..n} \neg(c'^i_i \in X_i) \)
From [7] and [12] and uniqueness of label \( c \)
on [69], we have
\( c \in X' \land \)
\( \bigvee_{i} c_i = c_i \land \)
\( \sigma(c'\text{cond}_{\pi}(c')) \land \)
\( \bigwedge_{i=1..n} \neg(c'^i_i \in X') \)
Now, we show that
\[ c \in X' \land \]
\[ \forall c_i, c' = c_i \land \]
\[ \sigma_{c'}(\text{cond}_{\pi}(c')) \land \]
\[ \wedge_{i=1..n} \neg(c'_i \in X') \]
\[ \Rightarrow \]
\[ c' \in X' \]
We assume that
\[ (71) \ c \in X' \land \]
\[ (72) \ \forall c_i, c' = c_i \land \]
\[ (73) \ \sigma_{c'}(\text{cond}_{\pi}(c')) \land \]
\[ (74) \ \wedge_{i=1..n} \neg(c'_i \in X') \]
We show that
\[ c' \in X' \]
From [7] and [12] on [71], we have
\[ \exists i \in \{0..n\} : \]
\[ (75) \ c \in X_i \]
From [7] and [12] on [74], we have
\[ \forall i \in \{0..n\} : \]
\[ (76) \ \wedge_{i=1..n} \neg(c'_i \in X_i) \]
By Lemma 15.4 on [65], [66], [11], [75], [72], [73] and [76], we have
\[ (77) \ c' \in X_i \]
From [7] and [12] on [77], we have
\[ c' \in X' \]
Case rule OCONTROL:
Similar to rule ICONTROL using Lemma 15.5.
Case rule TSEQ:
We have that
\[ (78) \ \mathcal{A} = l_1 < l_2 \lor l_2 < l_1 \lor l_1 = l_2 \]
\[ (79) \ \pi, \Gamma \vdash \text{exec}(l_1) \]
\[ (80) \ \pi, \Gamma \vdash \text{exec}(l_2) \]
\[ (81) \ \pi, \Gamma \vdash \text{thread}(l_1) = \text{thread}(l_2) \]
\[ (82) \ \pi, \Gamma \vdash \text{obj}(l_1) = \text{obj}(l_2) = \text{this} \lor \]
\[ (\neg \text{obj}(l_1) = \text{this} \land \neg \text{obj}(l_2) = \text{this}) \]
We show that
\[ \mathcal{X} \models l_1 < l_2 \lor l_2 < l_1 \lor l_1 = l_2 \]
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
\[ (83) \ \mathcal{X} \models \text{exec}(l_1) \]
\[ (84) \ \mathcal{X} \models \text{exec}(l_2) \]
\[ (85) \ \mathcal{X} \models \text{thread}(l_1) = \text{thread}(l_2) \]
\[ (86) \ \mathcal{X} \models \text{obj}(l_1) = \text{obj}(l_2) = \text{this} \lor \]
\[ (\neg \text{obj}(l_1) = \text{this} \land \neg \text{obj}(l_2) = \text{this}) \]
that is
\[ (87) \ l_1 \in X' \]
\[ (88) \ l_2 \in X' \]
\[ (89) \ \text{thread}_{X'}(l_1) = \text{thread}_{X'}(l_2) \]
\[ (90) \ \text{obj}_{X'}(l_1) = \text{obj}_{X'}(l_2) = \text{this} \lor \]
\[ (\neg \text{obj}_{X'}(l_1) = \text{this} \land \neg \text{obj}_{X'}(l_2) = \text{this}) \]
By [11] and [12] on [87] and [88], we have
\[ \exists i, j \in 0..n : \]
\[ (91) \ l_i \in X_i \land (X_i, \pi) \in [p_i] \]
\[ (92) \ l_j \in X_j \land (X_j, \pi) \in [p_i] \]
Case analysis on [90]:
Case
\[ (93) \ \text{obj}_{X'}(l_1) = \text{obj}_{X'}(l_2) = \text{this} \]
By Lemma 15.8 on [8], [87], [88], [93], we have
\[ \exists c_1, c_2 : \]
\[ (94) \ l_1 = c_1 \]
\[ (95) \ l_2 = c_2 \]
By Lemma 15.10 on [91], [92], [94], [95], we have
\[ (96) \ \text{thread}_{X'}(l_1) = T_i \]
\[ (97) \ \text{thread}_{X'}(l_2) = T_j \]
From [96], [97] and [89], we have
\[ (98) \ i = j \]
By Lemma 15.12 on [91], [92], and [94], [95], and [98], we have
\[ (99) \ l_i <_{X'} l_2 \lor l_2 <_{X'} l_1 \lor l_1 = l_2 \]
Case
\[ (100) \ \neg \text{obj}_{X'}(l_1) = \text{this} \land \]
\[ \neg \text{obj}_{X'}(l_2) = \text{this} \]
Similar to the previous case where lemmas 15.9, 15.11 and 15.13 are used.
Case rule TLOCAL:
We have that
\[ (101) \ \mathcal{A} = \text{thread}(l_1) = \text{thread}(l_2) \]
\[ (102) \ \mathcal{T}(\text{basename}(\phi)) = \text{ThreadLocal st} \]
\[ (103) \ \pi, \Gamma \vdash \text{exec}(l_1) \land \text{exec}(l_2) \]
\[ (104) \ \pi, \Gamma \vdash \text{obj}(l_1) = \text{obj}(l_2) = \phi[u] \]
We show that
\[ \mathcal{X} \models \text{thread}(l_1) = \text{thread}(l_2) \]
that is
\[ \text{thread}_{X'}(l_1) = \text{thread}_{X'}(l_2) \]
By the induction hypothesis on [104], and then [2], we have
\[ (105) \ \mathcal{X} \models \text{exec}(l_1) \land \text{exec}(l_2) \]
\[ (106) \ \mathcal{X} \models \text{obj}(l_1) = \text{obj}(l_2) = \phi[u] \]
that is
\[ (107) \ \text{obj}_{X'}(l_1) = \text{obj}_{X'}(l_2) = \phi[\sigma(u)] \]
\[ (108) \ l_1 \in X' \]
\[ (109) \ l_2 \in X' \]
From [107], we have
\[ (110) \ \text{basename}(\text{obj}_{X'}(l_1)) = \phi \]
\[ (111) \ \text{index}(\text{obj}_{X'}(l_1)) = \sigma(u) \]
\[ (112) \ \text{basename}(\text{obj}_{X'}(l_2)) = \phi \]
\[ (113) \ \text{index}(\text{obj}_{X'}(l_2)) = \sigma(u) \]
From Lemma 15.14 on [3], [102], [8], [108] and
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 If:

We have that
(119) \mathcal{A} = \text{obj}(c') = \xi_0 \land
\text{name}(c') = n \land
\text{thread}(c') = \xi_1 \land
\text{arg}(c') = \xi_2 \land
\text{retv}(c') = \xi_3

We show that
(120) \mathcal{X} \models \mathcal{A}

that is
\text{obj}_X(c') = \sigma(c_0) \land
\text{name}_X(c') = n \land
\text{thread}_X(c') = \sigma(c_1) \land
\text{arg}_X(c') = \sigma(c_2) \land
\text{retv}_X(c') = \sigma(c_3)

By the induction hypothesis on [125], and then [2], we have
(127) \mathcal{X} \models \text{exec}(c')

that is
(128) c' \in \mathcal{X}'

From [7] and [128], we have
(129) c' \in \mathcal{X}

From [12] and [129], we have
\exists i \in \{0..n\}:
(130) c' \in \mathcal{X}_i

From Lemma 15.1 on [11] and [130], we have
(131) \text{obj}_X(c') = \xi_0 \land
\text{name}_X(c') = n \land
\text{thread}_X(c') = \xi_1 \land
\text{arg}_X(c') = \xi_2 \land
\text{retv}_X(c') = \xi_3

Case rule Caller:

We have that
(133) \text{obj}_X(c') = \sigma(c_0) \land
\text{name}_X(c') = n \land
\text{thread}_X(c') = \sigma(c_1) \land
\text{arg}_X(c') = \sigma(c_2) \land
\text{retv}_X(c') = \sigma(c_3)

From [132], [7], we have
(134) \mathcal{A} =
\text{c'}t = \text{thread}(c) \land
\text{c'}x^* = \text{arg}(c) \land
\bigvee_{i=1..n}\text{exec}(c_i) \land \text{arg}_1(c_i) = \text{retv}(c)\) and
(135) \pi, \Gamma \vdash \text{exec}(c)

(136) \pi, \Gamma \vdash \text{obj}(c) = \text{this}

(137) \pi, \Gamma \vdash \text{name}(c) = n

(138) t\text{par}_n(c) = t \land \text{par}_1(c) = x

(139) \text{Returns}_n(c) = [c_t]

We show that
\mathcal{X} \models \mathcal{A}

that is
\sigma(c') = \text{thread}_X(c) \land
\sigma(c'_x) = \text{arg}_X(c) \land
\bigvee_{i=1..n}\text{exec}(c_i) \land \text{arg}_1(c_i) = \text{retv}_X(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') = \text{thread}_X(c) \land
(147) \sigma(c'_x) = \text{arg}_X(c) \land
(148) \bigvee_{i=1..n}\text{exec}(c_i) \land \text{arg}_1(c_i) = \text{retv}_X(c)

From [7] on [146], [147], and [148], we have
\sigma(c') = \text{thread}_X(c) \land
\sigma(c'_x) = \text{arg}_X(c) \land
\bigvee_{i=1..n}\text{exec}(c_i) \land \text{arg}_1(c_i) = \text{retv}_X(c)

Case rule Ret:
We have that
(149) \( tpar_\pi(n) = t \land par_\pi(n) = x \)
(150) \( c' \in \text{Returns}_\pi(n) \)
(151) \( \pi, \Gamma \vdash \text{exec}(c'c') \)

(152) \( \mathcal{A} = \text{exec}(c) \land \text{obj}(c) = \text{this} \land \text{name}(c) = n \land \text{thread}(c) = c't \land \text{arg}(c) = c'x' \land \text{ret}(c) = \text{arg}(c') \)

We show that
\( X \models \mathcal{A} \)
that is
\( c \in X' \land \text{obj}_X(c) = \text{this} \land \text{name}_X(c) = n \land \text{thread}_X(c) = \sigma(c't) \land \text{arg}_X(c) = \sigma(c'x') \land \text{ret}_X(c) = \text{arg}(c'c') \)

By induction hypothesis on \[151\], and then \[2\], \[6\] and \[7\], we have
\[153\] \( c'c' \in X' \)
From \[7\] and \[153\], we have
\[154\] \( c'c' \in X \)
From Lemma 15.17 on \[6\], \[49\], \[150\], and \[154\], we have
\[155\] \( c \in X \land \text{obj}_X(c) = \text{this} \land \text{name}_X(c) = n \land \text{thread}_X(c) = \sigma(c't) \land \text{arg}_X(c) = \sigma(c'x') \land \text{ret}_X(c) = \text{arg}(c'c') \)

From \[7\] on \[155]-[159], we have
\( c \in X' \land \text{obj}_X(c) = \text{this} \land \text{name}_X(c) = n \land \text{thread}_X(c) = \sigma(c't) \land \text{arg}_X(c) = \sigma(c'x') \land \text{ret}_X(c) = \text{arg}(c'c') \)

Case rule CALLEE:
Similar to rule RET

Case rule XASYM:
We have that
\( \pi, \Gamma \vdash l \prec l' \)
\( \mathcal{A} = \neg(l' < l) \land \neg(l' \sim l) \land \neg(l' = l) \)
We show that
\( X \models \mathcal{A} \)
that is
\( \neg(l' \prec_X l) \land \neg(l' \sim_X l) \land \neg(l' = l) \)
Straightforward from Lemma 10.1.

Case rule XTOTAL:

Case rule X2X:
Straightforward from Lemma 10.5.

Case rule LASYM:
We have that
\( \pi, \Gamma \vdash l \prec_o l' \)
\( \mathcal{A} = \neg(l' <_o l) \land \neg(l' = l) \)
We show that
\( X \models \mathcal{A} \)
Let
\( \pi, \Gamma \vdash l \prec_o l' \)
We need to show that
\( \neg(l' \prec_o l) \land \neg(l' = l) \)

Straightforward from Lemma 10.13.

Case rule LTOTAL:
We have that
\( \pi, \Gamma \vdash \text{exec}(l) \land \text{exec}(l') \)
\( \pi, \Gamma \vdash \text{obj}(l) = \text{obj}(l') = o \)
\( \mathcal{A} = (l \prec_o l') \lor (l' \prec_o l) \lor (l' = l) \)
We show that
\( X \models \mathcal{A} \)
From \[165\], let
\( \pi, \Gamma \vdash l \prec_o l' \)
We need to show that
\( \neg(l' \prec_o l) \lor (l' \prec_o l) \lor (l' = l) \)

By induction hypothesis on \[166\] and \[167\], and then \[2\], \[6\] and \[7\], we have
\( l \in X' \land \text{obj}_X(l) = \sigma(o) \land \text{obj}_X(l') = \sigma(o) \)

From \[165\], we have
\( \pi, \Gamma \vdash l \prec_o l' \land \text{obj}(l) = \sigma(o) \land \text{obj}(l') = \sigma(o) \)

Case rule L2X:
We have that
\( \pi, \Gamma \vdash l \prec_o l' \)
From [187], we have
(180) $O = L(\sigma(o))$

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

From [10] on [180], we have
(182) $(X'|\sigma(o), L(\sigma(o))) \in \mathbb{H}_{L}(\sigma(o))$

By Lemma 10.16 on [182] and [181], we have
(183) $l < X' \land l' < X'$
(184) $\mathcal{A} = l < l' \land l = l'$

We show that
$X \models A$

that is
$(192) l < X', l'$

By induction hypothesis on [183], [184], and [185], and then [2], [6] and [7], we have
(187) $T <_{X'} T'$
(188) $l < X'$
(189) $thread_{X'}(l) = T$
(190) $l' < X'$
(191) $thread_{X'}(l') = T$

From [189], we have
(192) $l < X'|T$

From [189], we have
(193) $l' < X'|T'$

From [187], we have
(194) $\forall T, T': X'|T <_{H} X'|T'$

Case rule XTRANS:
Straightforward from Lemma 10.2.

Case rule XXTRANS:
Straightforward from Lemma 10.3.

Case rule LTRANS:
Straightforward from Lemma 10.14.

Case rule TREAL:
We have that
(183) $\pi, \Gamma \vdash T < T'$
(184) $\pi, \Gamma \vdash \text{exec}(l) \land \text{thread}(l) = T$
(185) $\pi, \Gamma \vdash \text{exec}(l') \land \text{thread}(l') = T'$
(186) $\mathcal{A} = l < l' \land l = l'$

We show that
$X \models A$

that is
$(197) \mathcal{A} = \text{exec}(l) \land \text{exec}(l') \land \text{obj}(l) = \text{obj}(l') = o$

From [194], [192] and [193], we have
(198) $reg' = \sigma(reg)$

Let
(199) $Reg = L(reg')$

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

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

By the definition of isWriter on [197], we have
(202) $\mathcal{A} = \exists \mathcal{W}:
\text{isWriter}_{\mathcal{W}}(\ell_{\mathcal{W}}) \land \ell_{\mathcal{W}} \prec \ell_{\mathcal{W}} \land \forall \ell'_{\mathcal{W}}: \text{isWriter}_{\mathcal{W}}(\ell'_{\mathcal{W}}) \Rightarrow
(\ell'_{\mathcal{W}} \leq \ell_{\mathcal{W}} \lor \ell_{\mathcal{W}} \prec \ell'_{\mathcal{W}}) \land \text{retv}(\ell_{\mathcal{W}}) = \text{arg}1(\ell_{\mathcal{W}})$

We show that
$X \models A$

that is
$(195) T_{\text{base}}(\text{reg}) = \text{AtomicRegister}$
(196) $\pi, \Gamma \vdash \text{isRead}_{\text{reg}}(lR)$
(197) $\mathcal{A} = \exists \mathcal{W}:
\text{isWriter}_{\mathcal{W}}(\ell_{\mathcal{W}}) \land \ell_{\mathcal{W}} \prec \ell_{\mathcal{W}} \land \forall \ell'_{\mathcal{W}}: \text{isWriter}_{\mathcal{W}}(\ell'_{\mathcal{W}}) \Rightarrow
(\ell'_{\mathcal{W}} \leq \ell_{\mathcal{W}} \lor \ell_{\mathcal{W}} \prec \ell'_{\mathcal{W}}) \land \text{retv}(\ell_{\mathcal{W}}) = \text{arg}1(\ell_{\mathcal{W}})$
we have
\[ \exists l_W:\]
\[ \text{isXWrite}_{X',\text{reg},\text{reg}}(l_W) \land
\]
\[ l_W \leq_{\text{reg}} l_R \land
\]
\[ \forall l_W': \text{isXWrite}_{X',\text{reg},\text{reg}}(l_W') \Rightarrow
\]
\[ (l_W' \leq_{\text{reg}} l_W \lor l_R \leq_{\text{reg}} l_W') \land
\]
\[ \text{retX}_{X',\text{reg}}(l_R) = \text{argX}_{X',\text{reg}}(l_W)
\]

After simplification, we have
\[ \exists l_W:\]
\[ \text{isXWrite}_{X',\text{reg}}(l_W) \land
\]
\[ l_W \leq_{\text{reg}} l_R \land
\]
\[ \forall l_W': \text{isXWrite}_{X',\text{reg}}(l_W') \Rightarrow
\]
\[ (l_W' \leq_{\text{reg}} l_W \lor l_R \leq_{\text{reg}} l_W') \land
\]
\[ \text{retX}_{X}(l_R) = \text{argX}_{X}(l_W)
\]

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

Case rule CASRegRead:
By Lemma 10.24.

Case rule CASRegCAST:
By Lemma 10.25.

Case rule CASRegCASF:
By Lemma 10.25.

Case rule LOCK:
We have that
\[ \forall l_i:\]
\[ \text{base}(l_i) = \text{Lock}
\]
\[ \pi, \Gamma \vdash \text{isXOwnerRespecting}(l_i) \land
\]
\[ \pi, \Gamma \vdash \text{isLock}_{\pi, \Gamma}(l_i) \land
\]
\[ \pi, \Gamma \vdash l_i <_{\text{lo}} l_{ui}
\]
\[ \mathcal{A} = \exists l,u,l_i:
\]
\[ \text{isXUnlock}_{\pi, \Gamma}(l_i) \land
\]
\[ \text{threadX}(l_i) = \text{threadX}(l_i) \land
\]
\[ \text{isXLock}_{\pi}(l_i) \land
\]
\[ \text{threadX}(l_i) = \text{threadX}(l_i) \land
\]
\[ l_{ui} < l_{lo}
\]

Let
\[ \pi, \Gamma \vdash \text{base}(l_i) \land
\]
\[ \exists l,u,l_i:
\]
\[ \pi, \Gamma \vdash \text{isXOwnerRespecting}(l_i) \land
\]
\[ \pi, \Gamma \vdash \text{isLock}_{\pi, \Gamma}(l_i) \land
\]
\[ \pi, \Gamma \vdash l_i <_{\text{lo}} l_{ui}
\]
\[ \mathcal{A} = \exists l,u,l_i:
\]
\[ \text{isXUnlock}_{\pi, \Gamma}(l_i) \land
\]
\[ \text{threadX}(l_i) = \text{threadX}(l_i) \land
\]
\[ \text{isXLock}_{\pi}(l_i) \land
\]
\[ \text{threadX}(l_i) = \text{threadX}(l_i) \land
\]
\[ l_{ui} < l_{lo}
\]

By induction hypothesis on [208]-[211], and then [2], [6] and [7], we have
\[ \exists l_i:
\]
\[ \pi, \Gamma \vdash \text{isXOwnerRespecting}_{\pi, \Gamma}(X') \land
\]
\[ \pi, \Gamma \vdash \text{isXLock}_{\pi, \Gamma}(l_i) \land
\]
\[ \pi, \Gamma \vdash \text{isXUnlock}_{\pi, \Gamma}(l_{ui}) \land
\]
\[ l_i < l_{ui}
\]

From [216]-[219], we have
\[ \exists l_i:
\]
\[ \pi, \Gamma \vdash \text{isXOwnerRespecting}_{\pi, \Gamma}(X'|lo') \land
\]
\[ \pi, \Gamma \vdash \text{isXLock}_{\pi, \Gamma}(l_i) \land
\]
\[ \pi, \Gamma \vdash \text{isXUnlock}_{\pi, \Gamma}(l_{ui}) \land
\]
\[ l_i < l_{ui}
\]

From [207] and [213], we have
\[ \exists l_i:
\]
\[ \pi, \Gamma \vdash \text{isXOwnerRespecting}_{\pi, \Gamma}(X'|lo') \land
\]
\[ \pi, \Gamma \vdash \text{isXLock}_{\pi, \Gamma}(l_i) \land
\]
\[ \pi, \Gamma \vdash \text{isXUnlock}_{\pi, \Gamma}(l_{ui}) \land
\]
\[ l_i < l_{ui}
\]

From Lemma 10.29 on [224], and [220]-[223], we have
\[ \exists l_i:
\]
\[ \pi, \Gamma \vdash \text{isXOwnerRespecting}_{\pi, \Gamma}(X'|lo') \land
\]
\[ \pi, \Gamma \vdash \text{isXLock}_{\pi, \Gamma}(l_i) \land
\]
\[ \pi, \Gamma \vdash \text{isXUnlock}_{\pi, \Gamma}(l_{ui}) \land
\]
\[ l_i < l_{ui}
\]

From [225]-[229], we have
\[ \exists l_i:
\]
\[ \pi, \Gamma \vdash \text{isXOwnerRespecting}_{\pi, \Gamma}(X'|lo') \land
\]
\[ \pi, \Gamma \vdash \text{isXLock}_{\pi, \Gamma}(l_i) \land
\]
\[ \pi, \Gamma \vdash \text{isXUnlock}_{\pi, \Gamma}(l_{ui}) \land
\]
\[ l_i < l_{ui}
\]

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

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

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

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

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

Case rule SCounter:
By Lemma 10.41.
Case rule BasicSetContains:
By Lemma 10.43.

Case rule BasicSetAdd:
By Lemma 10.44.

Case rule BasicMapGet:
By Lemma 10.46.

Case rule BasicMapPut:
By Lemma 10.47.

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

Lemma 15.1.
∀p,X,σ,ζ,c′:
((X,σ) ∈ [p]) ∧ ζ c′ ∈ X)
⇒
(obj_X(ζ c′) = ζ obj_π(c′) ∧ thread_X(ζ c′) = ζ thread_π(c′) ∧ name_X(ζ c′) = name_π(c) ∧ arg1_X(ζ c′) = ζ arg1_π(c′) ∧ retv_X(ζ c′) = ζ retv_π(c′)).

Proof.
Structural induction on p:
(1) Case p = c ⇒ n_τ(u^∗)
Straightforward form definition [1].
(2) Case p = p_1; p_2
(3) Case p = if b p_1 else p_2
Straightforward form definition [12] and the induction hypothesis.

Lemma 15.2.
∀p,X,σ,ζ,c′:
((X,σ) ∈ [p]) ∧ ζ c′ ∈ X)
⇒
basename(obj_X(ζ c′)) = obj_π(c′) ∧ name_X(ζ c′) = name_π(c′).

Proof.
Structural induction on p:
(1) Case p = c ⇒ n_τ(u^∗)
Straightforward form definition [1] and the uniqueness of label c.
(2) Case p = p_1; p_2
(3) Case p = if b p_1 else p_2
Straightforward form definition [12] and the induction hypothesis. □

Lemma 15.3.
Let

Labels(name_π(c)) = {c_i}
PreReturns_π(c′) = {c_f}

∀p,X,σ,c,c′:
((X,σ) ∈ [p]) ∧ c c′ ∈ X)
⇒
c ∈ X ∧ \( \bigvee_{c_i} c_i c = c \sigma(c cond_π(c')) ∧ \bigwedge_{c_r} \neg(c r c_r c) ∈ X)\).

Proof.
Structural induction on p:
(1) Case p = c ⇒ n_τ(u^∗)
Straightforward form definition [1].
(2) Case p = p_1; p_2
Straightforward form definition [11], the induction hypothesis and the uniqueness of label c.
(3) Case p = if b p_1 else p_2
Straightforward form definition [12] and the induction hypothesis. □
Labels(name_\pi(c)) = \{c_i\}
PreReturns_\pi(c') = \{c_i\}
\forall p, X, \sigma, c, c':
((X, \sigma) \in \llbracket p \rrbracket \land c \in X \land \forall c_i c' = c_i \land \sigma(\text{cond}_\pi(c')) \land \forall c_i \neg(c' c_i \in X))
\Rightarrow c' c_i \in X.

Proof:

Structural induction on p:
(1) Case p = c \triangleright n_t(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
\Rightarrow \sigma(\text{cond}_\pi(c))
\iff c \in X.

Proof:

Structural induction on p:
(1) Case p = c \triangleright n_t(u^*):x
   Straightforward form definition [1]
(2) Case p = p_1; p_2

Lemma 15.6.

\forall \pi, X, \varsigma, c_1, c_2:
X \in \mathbb{H}(\pi) \land \varsigma' c_1 \in X \land \varsigma' c_2 \in X \land c_1 \rightarrow \pi c_2
\Rightarrow \varsigma' c_1 <_{X} \varsigma' c_2.

Proof:

Case analysis on c_1 \rightarrow \pi c_2
(1) Case: the initialization order
   Straightforward form definition [17] and [13].
   X = X_0 \cdot X'
(2) Case: the sequential order of the sequential programs \pi_i
   \forall c_i, c_j \in \{c_i\}:
   (c_i \rightarrow_n c_j) \land c' c_i \in X' \land c' c_j \in X'
   \Rightarrow c' c_i <_{X'} c' c_j

Proof:

Structural induction on \pi_i
(1) Case p = c \triangleright n_t(u^*):x
   Straightforward form definition [1]
(2) Case p = p_1; p_2
   Straightforward form definition [12] and the induction hypothesis.
(3) Case p = if b p_1 else p_2
   Straightforward form definition [12] and the induction hypothesis. \qed

67
Lemma 15.7.
\[\forall \pi, X, c, c': \]
\[X \in H(\pi) \land c'c' \in X \implies (\text{Ev}(c) <_X \text{Ev}(c') \land \text{rEv}(c') <_X \text{rEv}(c)).\]

Proof:
We have that
(1) \[X \in H(\pi)\]
(2) \[c'c' \in X\]
We show that
\[\text{Ev}(c) <_X \text{Ev}(c') \land \text{rEv}(c') <_X \text{rEv}(c)\]

From definition 17 and [13] on [1] and [2], we have
\[\exists X_i:\]
(3) \[(X_i, \sigma) \in [p_i]\]
(4) \[c'c' \in X_i\]
(5) \[X_i \in X\]
We show that
\[\text{Ev}(c) <_X \text{Ev}(c')\]

Lemma 15.8.
\[\forall \pi, X, \sigma, c:\]
\[X \in H(\pi) \land l \in X \land \text{obj}_X(l) = \text{this} \land \]
\[\exists c: l = c.\]

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

Lemma 15.9.
\[\forall \pi, X, \sigma, c:\]
\[X \in H(\pi) \land l \in X \land \neg\text{obj}_X(l) = \text{this} \land \]
\[\exists c, c': l = c'c'.\]

Proof. Similar to Lemma 15.8.

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

\( \forall \pi, T, D, P, p_0, \ldots, p_n, X, \sigma, c : \)

\( (\pi = (T, D, P) \land P = p_0, (p_1 \| p_2 \| \ldots \| p_n) \land (X, \sigma) \in [p_i] \land c' c' \in X \land \) \)

\( \Rightarrow \sigma(\text{thread}_X(c')) = i. \)

Proof.

By structural induction on \( p_i \), we have

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

Lemma 15.12.

\( \forall p, X, \sigma, c_1, c_2 : \)

\( (X, \sigma) \in [p] \land c_1 \in X \land c_2 \in X \land \)

\( \Rightarrow c_1 <_X c_2 \lor c_2 <_X c_1 \lor c_1 = c_2. \)

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

Lemma 15.13.

\( \forall p, X, \sigma, c_1, c_2, c_3, c_4 : \)

\( (X, \sigma) \in [p] \land c_1, c_2, c_3, c_4 \in X \land \)

\( \Rightarrow c_1 c_2 <_X c_3 c_4 \lor c_3 c_4 <_X c_1 c_2 \land c_1 c_2 = c_3 c_4. \)

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

Lemma 15.14.

\( \forall \pi, X, \phi, st : \)

\( \pi = (T, D, P) \land T(\phi) = \text{Threadlocal st} \land X \in \mathbb{H}(\pi) \land l \in X \land \) basename(obj\_X(l)) = \( \phi \)

\( \Rightarrow \text{thread}_X(l) = \text{index}(\text{obj}_X(l)). \)
Lemma 15.15.

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

We show that

\[ \exists \sigma \in \Pi \]

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.

From the well-formedness conditions, we have

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

Proof:

We have

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

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

\[ \exists X_i : \]

(6) \( l \in X_i \)
(7) \( (X_i, \sigma) \in [p_i] \)
(8) \( \text{basename}(\text{obj}_{X_i}(l)) = \phi \)
(9) \( X_i \subseteq X \)

We show that

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

Structural induction on \( p_i \):

(11) Case \( p_i = c \circ \eta \)

Form definition [1], we have

(12) \( l = c^c \)
(13) \( \text{index}(\text{object}_{X_i}(c^c)) = c^i_{\text{index}}(c) \)
(14) \( \text{thread}_{X_i}(c^c) = c^i_{\text{thread}}(c) \)

Lemma 15.15.

\( \forall \pi, X, \sigma, L, c, n, t, x : \)

\( (X, \sigma, L) \in [\pi] \)

\( \text{tpar}_n(n) = t \land \text{par}_1(n) = x \)

\( \text{Returns}_n(n) = \{ \pi \} \)

\( c \in X \)

\( \text{obj}_{X}(c) = \text{this} \)

\( \text{name}_{X}(c) = n \)

\[ \Rightarrow \]

\( \sigma(c^t) = \text{thread}_{X}(c) \land \sigma(c^x) = \text{arg}_{X}(c) \land \bigvee_{i=1}^{n} \) 

\( (c^c_i) \in X \land \text{arg}_{1}(c^c) = \text{ret}_{v}(c) \). 

Proof:

We have that

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

We show that

\( \sigma(c^t) = \sigma(\text{thread}_{X}(c)) \land \sigma(c^x) = \sigma(\text{arg}_{X}(c)) \land \bigvee_{i=1}^{n} \) 

\( (c^c_i) \in X \land \)
\[\sigma(\text{arg} 1_X(c'c)) = \sigma(\text{ret}u_X(c))\]

Structural induction on \(p_i\):

15. Case \(p_i = c \triangleright n_r(u')x\)

From the Well-formedness condition of specifications that

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

we have

\[\exists c_r \in \{\overline{c_r}\} : \sigma(c'\text{cond}_r(c_i))\]

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

\[\forall c_i \in \{\overline{c_i}\} :\]

\[c'c_i \in X' \Leftrightarrow (\sigma(c'\text{cond}_\pi(c_i)) \land \forall c_j \in \text{PreReturns}_\pi(c_i) \Rightarrow \neg c'c_j \in X')\]

and

Lemma 15.16.

\(\forall X, \sigma, c, n, t, u, x'\):

\((X, \sigma) \in [c \triangleright n_r(u):x]\)

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

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

\[\Rightarrow \ c' = c''.\]

Proof.

We have that

1. \((X, \sigma) \in [c \triangleright n_r(u):x]\)
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

Case

\(c' = c''\)

Obvious

Case

\(c' \in \text{PreReturns}_\pi(c'')\)

By definition [1] on [5], we have

\[\neg c' \in X\]

which is contradiction to [4].

Case

\(c'' \in \text{PreReturns}_\pi(c')\)

By definition [1] on [4], we have

\[\neg c' \in X\]

which is contradiction to [5].

\[\Box\]
Proof.

We have that

1. \((X, \sigma, L) \in \Pi\)
2. \(t_{par}(n) = t \land \text{par1}(n) = x\)
3. \(c' \in \text{Returns}_{\pi}(n)\)
4. \(c' \in X\)

We show that

5. \((X_i, \sigma) \in \Pi\)
6. \(c' \in X_i\)
7. \(X_i \subseteq X\)

We show that

8. \(c \in X_i\) ∧

From definition 13 on [1] and [4], we have

9. \(\text{obj}_{X_i}(c) = \textbf{this} \land \text{name}_{X_i}(c) = n \land\)
10. \(\sigma(\text{thread}_{X_i}(c)) = \sigma(c't) \land\)
11. \(\sigma(\text{arg}_{X_i}^*(c)) = \sigma(c'x^*) \land\)
12. \(\sigma(\text{ret}_{X_i}(c)) = \sigma(\text{arg1}_{X_i}(c'c'))\)

Structural induction on \(p_i\):

13. Case \(p_i = c \triangleright n, (u^*)\): 
   Straightforward form definition [1] and Lemma 15.16.
14. Case \(p_i = p'p''\)
   Straightforward form definition [11],
   the induction hypothesis and
   the uniqueness of label \(c\).
15. Case \(p = \textbf{if } b p_1 \textbf{ else } p_2\)
   Straightforward form definition [12] and
   the induction hypothesis.

From [11] on [8]-[12], we have

\(c \in X \land\)

9. \(\text{obj}_{X_i}(c) = \textbf{this} \land \text{name}_{X_i}(c) = n \land\)
10. \(\sigma(\text{thread}_{X_i}(c)) = \sigma(c't) \land\)
11. \(\sigma(\text{arg}_{X_i}^*(c)) = \sigma(c'x^*) \land\)
12. \(\sigma(\text{ret}_{X_i}(c)) = \sigma(\text{arg1}_{X_i}(c'c'))\)

\(\square\)
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.

XTTRANS:
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
\[(\pi, \Gamma \vdash isW_r_{reg}(\ell_W, \ell_R) \land isW_r_{reg}(\ell_W', \ell_R)) \Rightarrow (\pi, \Gamma \vdash \ell_W = \ell_W')\]

BREG’:
Derived from rule BREG and the following
\[\pi, \Gamma \vdash isSequential(\ell) \Rightarrow \pi, \Gamma \vdash \forall \ell: (isR_{ead}(\ell) \lor isW_{rite}(\ell)) \Rightarrow isR_{aceFree}(\ell)\]

TREG:
Derived from rule TLOCAL, rule TSEQ and rule BREG’.

CASREGREAD’:
Derived from rule CASREGREAD and the following
\[(\pi, \Gamma \vdash isCWriter_{reg}(\ell_W, \ell_R) \land isCWriter_{reg}(\ell_W', \ell_R)) \Rightarrow (\pi, \Gamma \vdash \ell_W = \ell_W')\]

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

\[
\begin{align*}
\text{Inits}(X) &= \{ l | l \in X \land \text{obj}_X(l) = \text{this} \land \text{name}_X(l) = \text{init} \} \\
\text{Reads}(X) &= \{ l | l \in X \land \text{obj}_X(l) = \text{this} \land \text{name}_X(l) = \text{read} \} \\
\text{Writes}(X) &= \{ l | l \in X \land \text{obj}_X(l) = \text{this} \land \text{name}_X(l) = \text{write} \} \\
\text{Commits}(X) &= \{ l | l \in X \land \text{obj}_X(l) = \text{this} \land \text{name}_X(l) = \text{commit} \} \\
\text{Committed}(X) &= \{ T | \exists l: l \in \text{Commits}(X) \land \text{thread}_X(l) = T \land \text{ret}_X(l) = \text{C} \} \\
\text{Aborted}(X) &= \{ T | \exists l: l \in X \land \text{obj}_X(l) = \text{this} \land \text{thread}_X(l) = T \land \text{ret}_X(l) = \text{A} \}
\end{align*}
\]

Lemma 15.18.
\[
\forall X, \sigma, c:
\begin{align*}
(X, \sigma) &\in \llbracket \text{trans}_j \rrbracket \land \ c \in X \Rightarrow \\
& (c \in \text{Inits}(X) \land c = \text{IL}_j) \lor \\
& (c \in \text{Reads}(X)) \lor \\
& (c \in \text{Writes}(X)) \lor \\
& (c \in \text{Commits}(X) \land c = \text{CL}_j) \land \\
& (\text{IL}_j \leq c) \land \\
& (\text{CL}_j \in X \Rightarrow c \leq \text{CL}_j)
\end{align*}
\]

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

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

Proof:
Case \( j = 0 \):

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

\( \forall c \in \text{Returns}_\pi(\text{init}): \text{arg}_\pi(c) = \text{ok} \)
∀c ∈ Returns_π(write): arg1_π(c) ≠ C

and that in every execution of the transaction trans_0, 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 ∈ Returns_π(init): arg1_π(c) = ok
∀c ∈ Returns_π(read): arg1_π(c) ≠ C
∀c ∈ Returns_π(write): arg1_π(c) ≠ C
∀c ∈ Returns_π(commit): arg1_π(c) = C ∨ arg1_π(c) = A.

Lemma 15.21.

∀π ∈ ΠTM: ∀X ∈ H(π): ∀T ∈ Trans(X): Let l = commitOf(T): l ∈ Inits(X) ∧ thread_X(l) = T

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

Lemma 15.22.

∀π ∈ ΠTM: ∀X ∈ H(π): ∀l, l’:

(l ∈ Inits(X) ∧ l’ ∈ Inits(X) ∧ thread_X(l) = thread_X(l’)) ⇒ l = l’


Lemma 15.23.

∀π ∈ ΠTM: ∀X ∈ H(π): ∀l, l’:

(l ∈ Inits(X) ∧ l’ ∈ X ∧ obj_X(l’) = this ∧ thread_X(l) = thread_X(l’)) ⇒ l ≤_X l’


Lemma 15.24.

∀π ∈ ΠTM: ∀X ∈ H(π): ∀T ∈ Trans(X)

Let l = commitOf(T):

T ∈ Committed(X) ⇒

(l ∈ Commit(X) ∧ thread_X(l) = T)

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

Lemma 15.25.

∀π ∈ ΠTM: ∀X ∈ H(π): ∀l, l’:

(l ∈ Commit(X) ∧ l’ ∈ Commit(X) ∧ thread_X(l) = thread_X(l’)) ⇒ l = l’

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


∀π ∈ ΠTM: ∀X ∈ H(π): ∀l, l’:

(l ∈ X ∧ obj_X(l) = this ∧ l’ ∈ Commit(X) ∧ thread_X(l) = thread_X(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 ∈ ΠTM: ∀X ∈ H(π): ∀t: 0 ≤ t ≤ n

(t ∈ Committed(X) ∧ t ∈ ¬Aborted(X)) ∨ (t ∈ Aborted(X) ∧ t ∈ ¬Committed(X))


Lemma 14.1

∀π ∈ ΠTM: π ⊨ Γ_0.

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