1 semantics

We model the state of a database by a time-stamp heap that is a partial function from locations to their histories. The history is a partial function from times to a set of events. A event is a triple consisting of the value being read or written, the operation, i.e. either read or write, and the transaction identifier. We use thread pool to model the concurrency. Each thread has a local stack and a local time, but a globally shared time-stamp heap. Therefore a thread pool is a partial functions from thread identifiers to the corresponding stack, time and transactions. The state of each transaction, i.e. local state, are a stack which is shared between transactions from the same thread, a heap that is a snapshot of the time-stamp heap, and fingerprints that are the heap locations being read and written.

$$
\begin{array}{rcl}
l \in \texttt{Loc} & \triangleq & \mathbb{N} \\
v \in \texttt{Val} & \triangleq & \mathbb{N} \cup \texttt{Loc} \\
\texttt{Var} & \triangleq & \{\texttt{x}, \texttt{y}, \ldots\} \\
t \in \texttt{TimeStamp} & \triangleq & \texttt{rational number or real number} \\
h \in \texttt{Heap} & \triangleq & \texttt{Loc} \rightharpoonup \texttt{Val} \\
s \in \texttt{Stack} & \triangleq & \texttt{Var} \rightarrow \texttt{Val} \\
o \in \texttt{Operation} & \triangleq & \{\texttt{r}, \texttt{w}, \texttt{s}, \texttt{e}\} \\
\mathcal{T} \subseteq \texttt{TransID} & \triangleq & \{\alpha, \beta, \ldots\} \\
\texttt{ThreadID} & \triangleq & \{i, j, \ldots\} \\
rs \in \texttt{ReadSet}, ws \in \texttt{WriteSet} & \triangleq & \mathcal{P}(\texttt{Loc}) \\
\hbar \in \texttt{TimeStampHeap} & \triangleq & \texttt{Loc} \rightharpoonup (\texttt{TimeStamp} \rightharpoonup (\texttt{Val} \times \texttt{Operation} \times \texttt{TransID})) \\
(s, \hbar, t) \in \texttt{ThreadState} & \triangleq & \texttt{Stack} \times \texttt{TimeStampHeap} \times \texttt{TimeStamp} \\
\eta \in \texttt{ThreadPool} & \triangleq & \texttt{ThreadID} \rightharpoonup \texttt{Stack} \times \texttt{TimeStamp} \times \mathbb{P} \\
\Sigma \in \texttt{State} & \triangleq & \texttt{TimeStampHeap} \times \texttt{ThreadPool} \\
\sigma = (s, h, rs, ws) \in \texttt{LocalState} & \triangleq & \texttt{Stack} \times \texttt{Heap} \times \texttt{ReadSet} \times \texttt{WriteSet}
\end{array}
$$

The arithmetic expression and boolean expression are standard and have no side effect.

$$\mathbb{E} ::= \ v \ \mid \ \texttt{x} \ \mid \ \mathbb{E} + \mathbb{E} \ \mid \ \mathbb{E} * \mathbb{E} \ \mid \ \ldots$$

$$
\begin{array}{rcl}
[\![v]\!]_s & \triangleq & v \\
[\![\texttt{x}]\!]_s & \triangleq & s(v) \\
[\![\mathbb{E}_1 + \mathbb{E}_2]\!]_s & \triangleq & [\![\mathbb{E}_1]\!]_s + [\![\mathbb{E}_2]\!]_s \\
[\![\mathbb{E}_1 * \mathbb{E}_2]\!]_s & \triangleq & [\![\mathbb{E}_1]\!]_s * [\![\mathbb{E}_2]\!]_s
\end{array}
$$

$$\mathbb{B} ::= \ \texttt{true} \ \mid \ \texttt{false} \ \mid \ \mathbb{E} = \mathbb{E} \ \mid \ \mathbb{E} < \mathbb{E} \ \mid \ \texttt{not}\ \mathbb{B} \ \mid \ \mathbb{B}\ \texttt{and}\ \mathbb{B} \ \mid \ \mathbb{B}\ \texttt{or}\ \mathbb{B} \ \mid \ \ldots$$

$$
\begin{array}{rcl}
[\![\texttt{true}]\!]_s & \triangleq & \texttt{true} \\
[\![\texttt{false}]\!]_s & \triangleq & \texttt{false} \\
[\![\mathbb{E}_1 = \mathbb{E}_2]\!]_s & \triangleq & [\![\mathbb{E}_1]\!]_s = [\![\mathbb{E}_2]\!]_s \\
[\![\mathbb{E}_1 < \mathbb{E}_2]\!]_s & \triangleq & [\![\mathbb{E}_1]\!]_s < [\![\mathbb{E}_2]\!]_s \\
[\![\texttt{not}\ \mathbb{B}]\!]_s & \triangleq & \neg\, [\![\mathbb{B}]\!]_s \\
[\![\mathbb{B}_1\ \texttt{and}\ \mathbb{B}_2]\!]_s & \triangleq & [\![\mathbb{B}_1]\!]_s \wedge [\![\mathbb{B}_2]\!]_s \\
[\![\mathbb{B}_1\ \texttt{or}\ \mathbb{B}_2]\!]_s & \triangleq & [\![\mathbb{B}_1]\!]_s \vee [\![\mathbb{B}_2]\!]_s
\end{array}
$$

$$\mathbb{C} ::= \begin{array}{l} \texttt{skip} \ \mid \ \texttt{x} := \mathbb{E} \ \mid \ [\mathbb{E}] := \mathbb{E} \ \mid \ \texttt{x} := [\mathbb{E}] \ \mid \ \texttt{if}\,(\mathbb{B})\ \mathbb{C}\ \texttt{else}\ \mathbb{C} \ \mid \\ \texttt{while}\,(\mathbb{B})\ \mathbb{C} \ \mid \ \mathbb{C}\,;\mathbb{C} \end{array}$$

The syntax and semantics of a single transaction are standard except *mutate* and *deref*. The *mutate* also adds the location being written to the write fingerprint set ws and the *deref* adds the location to rs. Note that there is no parallel composition, because it is within a transaction.

$$(-,-) \ \rightsquigarrow_l \ (-,-) \ \triangleq \ (\texttt{LocalState} \times \mathbb{C}) \times (\texttt{LocalState} \times \mathbb{C})$$

$$\frac{[\![\mathbb{E}]\!]_s = v}{(s, h, rs, ws), \texttt{x} := \mathbb{E} \ \rightsquigarrow_l \ (s[\texttt{x} \mapsto v], h, rs, ws), \texttt{skip}} \ ass$$

$$\frac{[\![\mathbb{E}_1]\!]_s = l \quad [\![\mathbb{E}_2]\!]_s = v \quad l \in \mathrm{dom}(h)}{(s, h, rs, ws), [\mathbb{E}_1] := \mathbb{E}_2 \ \rightsquigarrow_l \ (s, h[l \mapsto v], rs, ws \cup \{l\}), \texttt{skip}} \ mutate$$

$$\frac{[\![\mathbb{E}]\!]_s = l \quad v = h(l) \quad l \in \mathrm{dom}(h)}{(s,h,rs,ws), \mathtt{x} := [\mathbb{E}] \ \rightsquigarrow_l \ (s[\mathtt{x} \mapsto v], h, rs \cup \{l\}, ws), \mathtt{skip}}\ \mathit{deref}$$

$$\frac{[\![\mathbb{B}]\!]_s = \mathtt{true}}{(s,h,rs,ws), \mathtt{if}\ (\mathbb{B})\ \mathbb{C}_1\ \mathtt{else}\ \mathbb{C}_2 \ \rightsquigarrow_l \ (s,h,rs,ws), \mathbb{C}_1}\ \mathit{ifelsetrue}$$

$$\frac{[\![\mathbb{B}]\!]_s = \mathtt{false}}{(s,h,rs,ws), \mathtt{if}\ (\mathbb{B})\ \mathbb{C}_1\ \mathtt{else}\ \mathbb{C}_2 \ \rightsquigarrow_l \ (s,h,rs,ws), \mathbb{C}_2}\ \mathit{ifelsefalse}$$

$$\frac{[\![\mathbb{B}]\!]_s = \mathtt{true}}{(s,h,rs,ws), \mathtt{while}\,(\mathbb{B})\ \mathbb{C} \ \rightsquigarrow_l \ (s,h,rs,ws), \mathbb{C}\,;\mathtt{while}\,(\mathbb{B})\ \mathbb{C}}\ \mathit{whiletrue}$$

$$\frac{[\![\mathbb{B}]\!]_s = \mathtt{false}}{(s,h,rs,ws), \mathtt{while}\,(\mathbb{B})\ \mathbb{C} \ \rightsquigarrow_l \ (s,h,rs,ws), \mathtt{skip}}\ \mathit{whilefalse}$$

$$\frac{}{(s,h,rs,ws), \mathtt{skip}\,; \mathbb{C}_2 \ \rightsquigarrow_l \ (s,h,rs,ws), \mathbb{C}_2}\ \mathit{seqskip}$$

$$\frac{(s,h,rs,ws), \mathbb{C}_1 \ \rightsquigarrow_l \ (s',h',rs',ws'), \mathbb{C}_1'}{(s,h,rs,ws), \mathbb{C}_1\,; \mathbb{C}_2 \ \rightsquigarrow_l \ (s',h',rs',ws'), \mathbb{C}_1'\,; \mathbb{C}_2}\ \mathit{seqnonskip}$$

A program is sequential and parallel composition of transactions. To give semantics for a program, we extend the syntax by adding an extra waiting command, $\mathtt{wait}(i)$, as suffix. Intuitively, this $\mathtt{wait}(i)$ indicates that current thread is waiting another thread identified by i until it commits all its transactions and then join the thread.

$$\mathbb{P} ::= \ \mathtt{skip} \ \mid \ \left[\mathbb{C}\right] \ \mid \ \mathbb{P}\,;\mathbb{P} \ \mid \ \mathtt{if}\,(\mathbb{B})\ \mathbb{P}\ \mathtt{else}\ \mathbb{P} \ \mid \ \mathtt{while}\,(\mathbb{B})\ \mathbb{P} \ \mid \ \mathbb{P} \parallel \mathbb{P}$$

$$\mathbb{P}^{\uparrow} ::= \ \mathtt{wait}(i) \ \mid \ \mathbb{P} \ \mid \ \mathbb{P}^{\uparrow}\,;\mathtt{wait}(i) \ \mid$$

We will explain the *commit*, *par* and *wait*, and the rest are straightforward. We give label to each transition, and these labels are only usefully for parallel composition.

The *commit* rule says that a transaction prophesies a starting time when this transaction takes a snapshot h_s and runs locally, and an ending time when it successfully commits ensured by the `allowcommit`.

The *par* rule forks a new thread and appends a $\mathtt{wait}(i)$, parametrised by the new thread identifier i, at the merging point. The *wait* rule waits the thread i until it finishes, then joins the thread and updates the time to the maximum between the two threads. Note that these two rules are labelled with $\mathtt{fork}(i, \mathbb{P})$ or $\mathtt{join}(i, t)$ which are used by the semantics of a top level theadpool.

$$\begin{array}{rcl}
\iota \in \mathtt{Label} & \triangleq & \mathtt{id} \ \mid \ \mathtt{cmt}(\alpha) \ \mid \ \mathtt{fork}(i, \mathbb{P}) \ \mid \ \mathtt{join}(i, t) \\
(-,-) \ \overset{-}{\rightsquigarrow}_t \ (-,-) & \triangleq & (\mathtt{ThreadState} \times \mathbb{P}^{\uparrow}) \times \mathtt{Label} \times (\mathtt{ThreadState} \times \mathbb{P}^{\uparrow})
\end{array}$$

$$\begin{array}{rcl}
\mathit{startstate}(\hbar, t) & \triangleq & \lambda l.\, v \\
& & \mathtt{where}\ \exists t' \leq t.\, \hbar(l)(t') = (v, \mathtt{w}, -) \wedge \forall t'' \in (t', t).\, \hbar(l)(t'') = (-, \mathtt{w}, -) \\
\mathsf{allowcommit}(\hbar, ws, rs, t_s, t_e) & \triangleq & \mathsf{wellformhist}(\hbar, ws, rs, t_s, t_e) \wedge \mathsf{consistent}(\hbar, ws, rs, t_s, t_e) \\
\mathsf{wellformhist}(\hbar, ws, rs, t_s, t_e) & \triangleq & \forall t \in \{t_s, t_e\}, l \in ws \cup rs.\, \hbar(l)(t) \uparrow \\
\mathsf{consistent}(\hbar, ws, rs, t_s, t_e) & \triangleq & \forall l_w \in ws, t \in (t_s, t_e).\, \hbar(l_w)(t) \neq (-, \mathtt{w}, -) \wedge \\
& & \forall \alpha.\, \nexists t_{\alpha s} < t_e, t_{\alpha e} > t_e.\, \hbar(l_w)(t_{\alpha s}) = (-,-,\alpha) \wedge \hbar(l_w)(t_{\alpha e}) = (-, \mathtt{w}, \alpha) \wedge \\
& & \exists t_{min} = \min(\{t'' \mid t'' > t_e \wedge \hbar(l_w)(t'') \downarrow\}).\, \hbar(l_w)(t_{min}) \neq (-, \mathtt{r}, -) \\
\mathit{commitTrans}(\hbar, h_s, h_e, ws, rs, \alpha, t_s, t_e) & \triangleq & \lambda l. \begin{cases} \hbar(l) & l \notin ws \cup rs \\ \hbar(l)[t_s \mapsto (h_s(l), \mathtt{s}, \alpha)][t_e \mapsto (h_e(l), \mathtt{w}, \alpha)] & l \in ws \setminus rs \\ \hbar(l)[t_s \mapsto (h_s(l), \mathtt{r}, \alpha)][t_e \mapsto (h_e(l), \mathtt{e}, \alpha)] & l \in rs \setminus ws \\ \hbar(l)[t_s \mapsto (h_s(l), \mathtt{r}, \alpha)][t_e \mapsto (h_e(l), \mathtt{w}, \alpha)] & l \in rs \cap ws \end{cases} \\
\mathit{freshTransId}(\hbar) & \triangleq & \alpha\ \mathtt{where}\ \alpha \notin \left\{ \alpha' \ \middle|\ (-,-,\alpha') \in \bigcup_{l,t} \hbar(l)(t) \right\}
\end{array}$$

$$\frac{\begin{array}{c} t_s \geq t \quad t_e > t_s \quad h_s = startstate(\hbar, t_s) \quad (s, h_s, \emptyset, \emptyset), \mathbb{C} \rightsquigarrow_l^* (s', h_e, rs, ws), \texttt{skip} \\ \mathsf{allowcommit}(\hbar, ws, rs, t_s, t_e) \quad \alpha = freshTransId(\hbar) \quad \hbar' = commitTrans(\hbar, h_s, h_e, ws, rs, \alpha, t_s, t_e) \end{array}}{(s, \hbar, t), \left[\mathbb{C}\right] \overset{\texttt{cmt}(\alpha)}{\rightsquigarrow}_t (s', \hbar', t_e), \texttt{skip}} \ \textit{commit}$$

$$\frac{[\![\mathbb{B}]\!]_s = \texttt{true}}{(s, \hbar, t), \texttt{if}\,(\mathbb{B})\ \mathbb{P}_1\ \texttt{else}\ \mathbb{P}_2 \overset{\texttt{id}}{\rightsquigarrow}_t (s, \hbar, t), \mathbb{P}_1} \ \textit{conditiontrue}$$

$$\frac{[\![\mathbb{B}]\!]_s = \texttt{false}}{(s, \hbar, t), \texttt{if}\,(\mathbb{B})\ \mathbb{P}_1\ \texttt{else}\ \mathbb{P}_2 \overset{\texttt{id}}{\rightsquigarrow}_t (s, \hbar, t), \mathbb{P}_2} \ \textit{conditionfalse}$$

$$\frac{[\![\mathbb{B}]\!]_s = \texttt{false}}{(s, \hbar, t), \texttt{while}\,(\mathbb{B})\ \mathbb{P} \overset{\texttt{id}}{\rightsquigarrow}_t (s, \hbar, t), \texttt{skip}} \ \textit{norep}$$

$$\frac{[\![\mathbb{B}]\!]_s = \texttt{true}}{(s, \hbar, t), \texttt{while}\,(\mathbb{B})\ \mathbb{P} \overset{\texttt{id}}{\rightsquigarrow}_t (s, \hbar, t), \mathbb{P}\,;\texttt{while}\,(\mathbb{B})\ \mathbb{P}} \ \textit{rep}$$

$$\frac{}{(s, \hbar, t), \texttt{skip}\,; \mathbb{P}^{\uparrow} \overset{\texttt{id}}{\rightsquigarrow}_t (s, \hbar, t), \mathbb{P}^{\uparrow}} \ \textit{seqskip}$$

$$\frac{(s, \hbar, t), \mathbb{P}_1^{\uparrow} \overset{\iota}{\rightsquigarrow}_t (s', \hbar', t'), \mathbb{P}_1^{\uparrow\prime}}{(s, \hbar, t), \mathbb{P}_1^{\uparrow}\,; \mathbb{P}_2^{\uparrow} \overset{\iota}{\rightsquigarrow}_t (s', \hbar', t'), \mathbb{P}_1^{\uparrow\prime}\,; \mathbb{P}_2^{\uparrow}} \ \textit{seqnoskip}$$

$$\frac{}{(s, \hbar, t), \mathbb{P}_1 \parallel \mathbb{P}_2 \overset{\texttt{fork}(i, \mathbb{P}_2)}{\rightsquigarrow}_t (s, \hbar, t), \mathbb{P}_1\,; \texttt{wait}(i)} \ \textit{par}$$

$$\frac{}{(s, \hbar, t), \texttt{wait}(i) \overset{\texttt{join}(i, t')}{\rightsquigarrow}_t (s, \hbar, \max\{t, t'\}), \texttt{skip}} \ \textit{wait}$$

$$- \overset{-}{\rightsquigarrow}_g - \ \triangleq\ \texttt{State} \times \texttt{Label} \times \texttt{State}$$

The semantics of theadpool picks a thread to run one step. If the step is a fork, it generates a new thread with a new stack and a local time that is the same as its parent thread. If it is a join, the threadpool passes the child's local time to its parent thread.

$$\frac{(s, \hbar, t), \mathbb{P}^{\uparrow} \overset{\iota}{\rightsquigarrow}_t (s', \hbar', t'), \mathbb{P}^{\uparrow\prime} \quad \iota \in \{\texttt{id}, \texttt{cmt}(-)\}}{(\hbar, \eta \uplus \left\{i \mapsto (s, t, \mathbb{P}^{\uparrow})\right\}) \overset{\iota}{\rightsquigarrow}_g (\hbar', \eta \uplus \left\{i \mapsto (s', t', \mathbb{P}^{\uparrow\prime})\right\})} \ \textit{single}$$

$$\frac{(s, \hbar, t), \mathbb{P}^{\uparrow} \overset{\texttt{fork}(i', \mathbb{P}'')}{\rightsquigarrow}_t (s', \hbar', t'), \mathbb{P}^{\uparrow\prime}}{(\hbar, \eta \uplus \left\{i \mapsto (s, t, \mathbb{P}^{\uparrow})\right\}) \overset{\texttt{fork}(i', \mathbb{P}'')}{\rightsquigarrow}_g (\hbar', \eta \uplus \left\{i \mapsto (s', t', \mathbb{P}^{\uparrow\prime}), i' \mapsto (\lambda \texttt{x}.\, 0, t', \mathbb{P}'')\right\})} \ \textit{par}$$

$$\frac{(s, \hbar, t), \mathbb{P}^{\uparrow} \overset{\texttt{join}(i', t'')}{\rightsquigarrow}_t (s', \hbar', t'), \mathbb{P}^{\uparrow\prime}}{(\hbar, \eta \uplus \left\{i \mapsto (s, t, \mathbb{P}^{\uparrow}), i' \mapsto (s', t'', \texttt{skip})\right\}) \overset{\texttt{join}(i', t'')}{\rightsquigarrow}_g (\hbar', \eta \uplus \left\{i \mapsto (s', t', \mathbb{P}^{\uparrow\prime})\right\})} \ \textit{wait}$$

A program to check

$$\left[\begin{array}{l} \texttt{x} := [l_x]; \\ \texttt{if}\,(\texttt{x} = 0) \\ \quad [l_y] := 1 \end{array}\right] \parallel \left[\begin{array}{l} \texttt{y} := [l_y]; \\ \texttt{if}\,(\texttt{x} = 0) \\ \quad [l_x] := 1 \end{array}\right] \parallel \left[[l_x] := 2\right] \parallel \left[[l_y] := 2\right]$$

2 Proof of semantics

Lemma 2.1. A history cannot be overwritten, i.e. $\forall \hbar, \hbar', l, t.\, (\hbar, -) \overset{-}{\rightsquigarrow}_g (\hbar', -) \implies \hbar(l)(t) \subseteq \hbar'(l)(t)$

Proof. Induction on the semantics. Except the *commit*, the rest is trivial. From the `wellformhist`, a new transition must pick a starting time and a ending time that have no event for those locations touched. □

Lemma 2.2. All the reads of a transaction happen before all the writes. This is $\forall \hbar, l, l', t, t', \alpha.\, \hbar(l)(t) = (-, \mathtt{r}, \alpha) \wedge \hbar(l')(t') = (-, \mathtt{w}, \alpha) \implies t < t'$.

Proof. From the *commit* that $t_s < t_e$. □

Lemma 2.3. A transaction's reads and starts operations among all locations happen in the same time, so do all the writes and ends operations. This is $\forall \hbar, l, l', t, t', \alpha, o, o'.\, \hbar(l)(t)(-, o, \alpha) \wedge \hbar(l')(t') = (-, o', \alpha) \wedge (o, o' \in \{\mathtt{s}, \mathtt{r}\} \vee o, o' \in \{\mathtt{e}, \mathtt{w}\}) \implies t = t'$.

Proof. Given Lemma 2.1, induction on semantics. The *commit* is by the *commitTrans*, the rest is trivial. □

Definition 2.4. Session order `so`, or program order.

$$
\begin{array}{rcl}
lc \in \mathtt{LastCommit} & \triangleq & \mathtt{ThreadID} \rightharpoonup \mathcal{P}(\mathtt{TransID}) \\
\mathit{sessionOrder}^0(\hbar_{init}, \eta_{init}) & \triangleq & \{(\emptyset, \emptyset, \hbar_{init}, \eta_{init}, \emptyset)\} \\
\mathit{sessionOrder}^n(\hbar_{init}, \eta_{init}) & \triangleq & \left\{ (\mathcal{T}, \mathtt{so}, \hbar, \eta \uplus \{i \mapsto -\}, lc) \middle|
\begin{array}{l}
\exists \mathcal{T}', \mathtt{so}', \hbar', \eta', lc', \iota. \\
(\mathcal{T}', \mathtt{so}', \hbar', \eta' \uplus \{i \mapsto -\}, lc') \in \\
\qquad \mathit{sessionOrder}^{n-1}(\eta_{init}, \hbar_{init}) \wedge \\
(\hbar', \eta' \uplus \{i \mapsto -\}) \overset{\iota}{\rightsquigarrow}_g (\hbar, \eta \uplus \{i \mapsto -\}) \wedge \\
\iota = \mathtt{id} \implies \\
\big(\mathcal{T} = \mathcal{T}' \wedge \mathtt{so} = \mathtt{so}' \wedge \eta = \eta' \wedge lc = lc'\big) \wedge \\
\exists \alpha.\, \iota = \mathtt{cmt}(\alpha) \implies \\
\begin{pmatrix}
\mathcal{T} = \mathcal{T}' \uplus \{\alpha\} \wedge \\
\mathtt{so} = \mathtt{so}' \uplus \{(\alpha', \alpha) | \alpha' \in lc'(i)\} \wedge \\
\eta = \eta' \wedge lc = lc'[i \mapsto \{\alpha\}]
\end{pmatrix} \wedge \\
\exists i'.\, \iota = \mathtt{fork}(i', -) \implies \\
\begin{pmatrix}
\mathcal{T} = \mathcal{T}' \wedge \mathtt{so} = \mathtt{so}' \wedge \eta = \eta' \uplus \{i' \mapsto -\} \wedge \\
lc = lc' \uplus \{i' \mapsto lc'(i)\}
\end{pmatrix} \wedge \\
\exists i''.\, \iota = \mathtt{join}(i'', -) \implies \\
\begin{pmatrix}
\mathcal{T} = \mathcal{T}' \wedge \mathtt{so} = \mathtt{so}' \wedge \eta = \eta' \uplus \{i'' \mapsto -\} \wedge \\
lc = lc'[i \mapsto lc'(i) \uplus lc'(i'')] \setminus \{i'' \mapsto -\}
\end{pmatrix} \wedge
\end{array}
\right\} \\
\mathit{histories}(\hbar_{init}, \eta_{init}) & \triangleq & \left\{ (\mathcal{T}, \mathtt{so}, \hbar) \middle| (\mathcal{T}, \mathtt{so}, \hbar, -, -) \in \bigcup_{n \in \mathbb{N}} \mathit{sessionOrder}^n(\hbar_{init}, \eta_{init}) \right\}
\end{array}
$$

Definition 2.5. Visibility and potential arbitration relations.

$$
\begin{array}{rcl}
\mathit{graph}(\mathcal{T}, \mathtt{so}, \hbar) & \triangleq & (\mathcal{T}, \mathtt{so}, \mathtt{vis}, \mathtt{tar}) \\
& & \mathtt{where\ vis} \triangleq \left\{ (\alpha, \alpha') \in \mathcal{T} \middle| \begin{array}{l} \exists l, l', t, t', o \in \{\mathtt{w}, \mathtt{e}\}, o' \in \{\mathtt{r}, \mathtt{s}\}.\, t < t' \wedge \\ \hbar(l)(t) = (-, o, \alpha) \wedge \hbar(l')(t') = (-, o', \alpha') \end{array} \right\} \\
& & \qquad \mathtt{tar} \triangleq \left\{ (\alpha, \alpha') \in \mathcal{T} \middle| \begin{array}{l} \exists l, l', t, t', o, o' \in \{\mathtt{w}, \mathtt{e}\}.\, t < t' \wedge \\ \hbar(l)(t) = (-, o, \alpha) \wedge \hbar(l')(t') = (-, o', \alpha') \end{array} \right\}
\end{array}
$$

Lemma 2.6. The read/start operations of all the needed locations happen in the same time, so do write/end operations. This is $\forall \hbar, \alpha, l, l', t, t', o, o'.\, (o, o' \in \{\mathtt{r}, \mathtt{s}\} \vee o, o' \in \{\mathtt{w}, \mathtt{e}\}) \wedge \hbar(l)(t) = (-, o, \alpha) \wedge \hbar(l')(t') = (-, o', \alpha) \implies t = t'$

Proof. Derive from *commitTrans* from the semantics. □

Lemma 2.7 (Session)**.** $\mathtt{so} \subseteq \mathtt{vis}$, for some $\mathcal{T}$ and $\hbar$, such that $\mathit{graph}(\mathcal{T}, \mathtt{so}, \hbar) = (\mathcal{T}, \mathtt{so}, \mathtt{vis}, -)$.

Proof. Given the definition of *sessionOrder*, assume a sequence of $\mathtt{so}_n$, each of which represents the result of the n-steps *sessionOrder*, so $\mathtt{so} = \bigcup_n \mathtt{so}_n \wedge \forall n, n'.\, n < n' \implies \mathtt{so}_n \subseteq \mathtt{so}_{n'}$. Therefore, if $(\alpha, \alpha') \in \mathtt{so}$, there must exist a n that $(\alpha, \alpha') \notin \mathtt{so}_{n-1}$ but $(\alpha, \alpha') \in \mathtt{so}_n$. This means there exists lc for same thread i, such that $\alpha \in lc(i)$. Since the lc is a partial function that records the last committed transactions of each thread. Note that if there is a new thread, it records the last committed transactions of its parent thread, while if it is a join point, it records both the parent and child last committed transactions. Then if the new transaction α' commits, it must pick a starting time greater than any transactions included in $lc(i)$. Assuming the time-stamp heap $\hbar$ after the new transaction α' committing and by the Lemma 2.6, we have $\forall l, l', t, t', o \in \{\mathtt{w}, \mathtt{e}\}, o' \in \{\mathtt{r}, \mathtt{s}\}.\, \hbar(l)(t) = (-, o, \alpha) \wedge \hbar(l')(t') = (-, o', \alpha') \implies t < t'$, which implies $(\alpha, \alpha') \in \mathtt{so}$. □

Lemma 2.8 (prefix)**.** $\mathtt{tar};\mathtt{vis} \subseteq \mathtt{vis}$, for some $\mathcal{T}$, $\mathtt{so}$ and $\hbar$, such that $graph(\mathcal{T}, \mathtt{so}, \hbar) = (\mathcal{T}, \mathtt{so}, \mathtt{vis}, \mathtt{tar})$.

Proof. For all $\alpha, \alpha', \alpha''$, if $(\alpha, \alpha') \in \mathtt{tar}$ and $(\alpha', \alpha'') \in \mathtt{vis}$, by the definitions of $\mathtt{so}$ and $\mathtt{vis}$, the commit time of α' is greater than the one of α but smaller than the start time of α''. Thus there must exist $l, l'', t, t', t'', o \in \{\mathtt{w}, \mathtt{e}\}$ and $o'' \in \{\mathtt{r}, \mathtt{s}\}$ such that $\hbar(l)(t) = (-, o, \alpha)$, $\hbar(l'')(t'') = (-, o'', \alpha'')$ and $t < t''$, thus $(\alpha, \alpha'') \in \mathtt{vis}$. □

Lemma 2.9 (nocoflict)**.** Two transactions cannot concurrently write to the same location, this means that one must observe another one. This is $\forall l, \alpha, \alpha'. \hbar(l)(-) = (-, \mathtt{w}, \alpha) \wedge \hbar(l)(-) = (-, \mathtt{w}, \alpha') \implies ((\alpha, \alpha') \in \mathtt{vis} \vee (\alpha', \alpha) \in \mathtt{vis}))$, for some $\mathcal{T}$, $\mathtt{so}$ and $\hbar$, such that $graph(\mathcal{T}, \mathtt{so}, \hbar) = (\mathcal{T}, \mathtt{so}, \mathtt{vis}, -)$.

Proof. Prove by contradiction. Assume $(\alpha, \alpha') \notin \mathtt{vis} \wedge (\alpha', \alpha) \notin \mathtt{vis}$, this intuitively means one transaction is overlapped with another. Let t_s, t_e, t'_s and t'_e be the start time and end time of transaction α and α' respectively. Because of the symmetric, we can assume that the start time of α is in between α', witch means $t'_s < t_s < t'_e$. Now we consider t_e. First note that $t_e > t_s$ by Lemma 2.2, therefore we need to consider two cases $t'_s < t_s < t_e < t'_e$ and $t'_s < t_s < t'_e < t_e$. Since both transaction write the same location l, those two cases violate the $\mathsf{consist}()$ requirement in the semantics, so one of the transactions must pick another start and end time. □

Lemma 2.10 (ext)**.** A transaction should read the last values it can observe. This means that for all transaction α and heap location l, if the transaction read a value v from the location, i.e. $\exists t. \hbar(l)(t) = (v, \mathtt{r}, \alpha)$, the last transaction α' who writes to the same location and can be observe by α, i.e. $(\alpha, \alpha') \in \mathtt{vis}$, should have written the same value, meaning $\exists v', t'. \hbar(l)(t') = (v', \mathtt{w}, \alpha') \implies v' = v$

Proof. Given the definition of $\mathtt{vis}$, we have $t' < t$. Because transaction α' is the last one who write to the location, this means that $\forall \alpha'', t'', o''. \hbar(l)(t'') = (-, o'', \alpha'') \implies o'' \neq \mathtt{w}$. Thus by the *startstate* in the semantics, we have $v' = v$.

Lemma 2.11 (acyclic)**.** Both $\mathtt{vis}$ and $\mathtt{tar}$ are acyclic.

Proof. Proof by contradiction. Assume there are transactions from α_0 to α_n for some n that form a circle by $\mathtt{vis}$ relation. By Lemma 2.3, let t^s_i and t^e_i be the start time and end time of the transaction α_i respectively. By Lemma 2.2, we have $t^s_i < t^e_i$, and by definition of $\mathtt{vis}$, thus $t^e_i < t^s_{(i+1) \bmod n}$. Therefore, $t^s_0 < t^e_0 < t^s_1 < \cdots < t^e_n < t^s_0$, so we have contradiction.

Similarly for $\mathtt{tar}$, we have contradiction that $t^e_0 < t^e_1 < \cdots < t^e_n < t^e_0$. □

Lemma 2.12 (totalorder)**.** The arbitration relations $\mathtt{tar}$ can be extended to a total order $\mathtt{ar}$ that does not violate Lemma 2.8, Lemma 2.10 and Lemma 2.11.

Proof. We construct $\mathtt{ar}$ from $\mathtt{tar}$. Assume there is an initialisation transaction α_{init} that happens before all other transactions, which means $\forall \alpha \in \mathcal{T}. (\alpha_{init}, \alpha) \in \mathtt{tar}$. Now we extend $\mathtt{tar}$ until it is a total order. Let $\mathtt{tar}_0 = \mathtt{tar}$, and clearly $\mathtt{tar}_0$ satisfies those Lemmas.

Now assume that we have $\mathtt{tar}_n$ for some n that satisfies Lemma 2.8, Lemma 2.10 and Lemma 2.11. We pick first two transactions α_1 and α_2 that satisfies the following. First, if a transaction can reach α_1, it also can reach α_2, and vice versa, $\forall \alpha \in \mathcal{T} \uplus \{\alpha_{init}\}. (\alpha, \alpha_1) \in \mathtt{tar}_n \iff (\alpha, \alpha_2) \in \mathtt{tar}_n$. Second, let $\mathcal{S}$ be the set of such transactions, we requires it is a strict total order with respect to $\mathtt{tar}_n$, i.e. $\forall \alpha, \alpha' \in \mathcal{S}. (\alpha, \alpha') \in \mathtt{tar}_n \vee (\alpha', \alpha) \in \mathtt{tar}_n$. Intuitively, α_1 and α_2 is a branching point since α_{init} with respect to $\mathtt{tar}_n$.

Immediately, there are two important properties of the transactions. Since that each step we add a new relations, so $\mathtt{vis} \subseteq \mathtt{tar} \subseteq \mathtt{tar}_0 \subseteq \cdots \subseteq \mathtt{tar}_n$. Therefore, given $(\alpha_1, \alpha_2) \notin \mathtt{tar} \wedge (\alpha_2, \alpha_1) \notin \mathtt{tar}$ and Lemma 2.3, these two transactions write locations in exactly the same time and given $(\alpha_1, \alpha_2) \notin \mathtt{vis} \wedge (\alpha_2, \alpha_1) \notin \mathtt{vis}$ and the contrapositive of Lemma 2.9, the transactions must write different locations. Now we claim that we can extend $\mathtt{tar}_n$ by arbitrarily adding a relation between the two transactions and then taking the transitive closure, for instance $\mathtt{tar}_{n+1} = (\mathtt{tar}_n \uplus \{(\alpha_1, \alpha_2)\})^+$, and this new $\mathtt{tar}_{n+1}$ still satisfies Lemma 2.8, Lemma 2.10 and Lemma 2.11.

Lemma 2.8 holds, because α_1 and α_2 commit at the same time, so $\forall \alpha. (\alpha_1, \alpha) \in \mathtt{vis} \iff (\alpha_2, \alpha) \in \mathtt{vis}$. Then, by the assumption $\mathtt{tar}_n; \mathtt{vis} \subseteq \mathtt{vis}$, we have $\mathtt{tar}_{n+1}; \mathtt{vis} \subseteq \mathtt{vis}$.

Lemma 2.10 holds because α_1 and α_2 write different locations. Therefore, even though there is a new relation (α_1, α_2), the transaction α_2 still reads the newest values it can observe. Similarly for those transactions who depends on the values written by α_1, they still read the newest values they can observe.

Note that $\mathtt{vis}$ remains the same some we only need to check $\mathtt{tar}_{n+1}$. Assume that the new relation intrudes a circle, this circle must contain α_1 and α_2, for instance $\alpha, \ldots \alpha_1, \alpha_2, \ldots \alpha'$. Given how we pick α_1 and α_2, all the transactions before α_1 in the circle also can reach α_2. This means $\alpha, \ldots \alpha_1, \ldots \alpha'$ is also a circle, which contradict with assumption that $\mathtt{tar}_n$ has no circle. □

3 blablabla

$$
\begin{array}{rcl}
P, Q \in \texttt{GlobalAssertion} & \triangleq & \begin{array}{l} \texttt{False} \mid \texttt{True} \mid \texttt{emp} \mid x \mapsto v \mid P * Q \mid \\ P \wedge Q \mid P \vee Q \mid \exists x.\, P \mid P \implies Q \end{array}
\end{array}
$$

$$
\begin{array}{rcl}
p, q \in \texttt{LocalAssertion} & \triangleq & \begin{array}{l} \texttt{False} \mid \texttt{True} \mid \texttt{emp} \mid x \mapsto_w v \mid x \mapsto_r v \mid p * q \mid \\ p \wedge q \mid p \vee q \mid \exists x.\, p \mid p \implies q \end{array}
\end{array}
$$

$$
\begin{array}{rcl}
a \in \texttt{Action} & \triangleq & \{p \leadsto q \mid p, q \in \texttt{LocalAssertion} \wedge p \texttt{ has no write tag}\} \\
a \in \texttt{Action} & \triangleq & p \leadsto q \mid a; \texttt{Action}
\end{array}
$$

Logic expression

$$[\![e]\!]_\iota$$

Assume there are special logical variables, ranging $l_1, l_2, \ldots$, that point to the corresponding locations in the global heap.

$$
\begin{array}{rcl}
[\![\texttt{False}]\!]_\iota & \triangleq & \emptyset \\
[\![\texttt{True}]\!]_\iota & \triangleq & \texttt{Heap} \\
[\![x \mapsto v]\!]_\iota & \triangleq & \{h \mid \exists t.\, h([\![x]\!]_\iota) = ([\![v]\!]_\iota, t)\} \\
[\![P * Q]\!]_\iota & \triangleq & \{h_P \uplus h_Q \mid h_P \in [\![P]\!]_\iota \wedge h_Q \in [\![Q]\!]_\iota\} \\
[\![P \wedge Q]\!]_\iota & \triangleq & \{h \mid h \in [\![P]\!]_\iota \wedge h \in [\![Q]\!]_\iota\} \\
[\![P \vee Q]\!]_\iota & \triangleq & \{h \mid h \in [\![P]\!]_\iota \vee h \in [\![Q]\!]_\iota\} \\
[\![\exists x.\, P]\!]_\iota & \triangleq & \left\{h \mid \exists v.\, h \in [\![P]\!]_{\iota[x \mapsto v]}\right\} \\
[\![P \implies Q]\!]_\iota & \triangleq & \{h \mid h \in [\![P]\!]_\iota \implies h \in [\![Q]\!]_\iota\}
\end{array}
$$

$$
\begin{array}{rcl}
[\![\texttt{False}]\!]_{s,\iota} & \triangleq & \emptyset \\
[\![\texttt{True}]\!]_{s,\iota} & \triangleq & \texttt{Heap} \times \texttt{Heap} \\
[\![\texttt{emp}]\!]_{s,\iota} & \triangleq & \{(\emptyset, \emptyset)\} \\
[\![x \mapsto_r v]\!]_{s,\iota} & \triangleq & \left\{(h_s, h_e) \mid \exists t.\, h_s([\![x]\!]_{s,\iota}) = ([\![v]\!]_{s,\iota}, t) \wedge h_e = ([\![v]\!]_{s,\iota}, t)\right\} \\
[\![x \mapsto_w v]\!]_{s,\iota} & \triangleq & \left\{(h_s, h_e) \mid \exists v_s, t_s, t_e.\, h_s([\![x]\!]_{s,\iota}) = (v_s, t_s) \wedge h_e = ([\![v]\!]_{s,\iota}, t_e) \wedge t_s < t_e\right\} \\
[\![p * q]\!]_{s,\iota} & \triangleq & \left\{(h_{ps} \uplus h_{qs}, h_{pe} \uplus h_{qe}) \mid (h_{ps}, h_{pe}) \in [\![p]\!]_{s,\iota} \wedge (h_{qs}, h_{qe}) \in [\![q]\!]_{s,\iota}\right\} \\
[\![p \wedge q]\!]_{s,\iota} & \triangleq & \left\{(h_s, h_e) \mid (h_s, h_e) \in [\![p]\!]_{s,\iota} \wedge (h_s, h_e) \in [\![q]\!]_{s,\iota}\right\} \\
[\![p \vee q]\!]_{s,\iota} & \triangleq & \left\{(h_s, h_e) \mid (h_s, h_e) \in [\![p]\!]_{s,\iota} \vee (h_s, h_e) \in [\![q]\!]_{s,\iota}\right\} \\
[\![\exists x.\, p]\!]_{s,\iota} & \triangleq & \left\{(h_s, h_e) \mid \exists v.\, (h_s, h_e) \in [\![p]\!]_{s,\iota[x \mapsto v]}\right\} \\
[\![p \implies q]\!]_{s,\iota} & \triangleq & \left\{(h_s, h_e) \mid (h_s, h_e) \in [\![p]\!]_{s,\iota} \implies (h_s, h_e) \in [\![q]\!]_{s,\iota}\right\}
\end{array}
$$

$$
\begin{array}{rcl}
\texttt{True} \mathbin{\hat{*}} p & \triangleq & p \\
\texttt{False} \mathbin{\hat{*}} - & \triangleq & \texttt{False} \\
x \mapsto_r v \mathbin{\hat{*}} x \mapsto_r v & \triangleq & x \mapsto_r v \\
x \mapsto_r - \mathbin{\hat{*}} x \mapsto_w v & \triangleq & x \mapsto_w v \\
(p_1 * q_1) \mathbin{\hat{*}} (p_2 * q_2) & \triangleq & (p_1 \mathbin{\hat{*}} p_2) * (q_1 \mathbin{\hat{*}} q_2) \\
(p \wedge q) \mathbin{\hat{*}} r & \triangleq & (p \mathbin{\hat{*}} r) \wedge (q \mathbin{\hat{*}} r) \\
(p \vee q) \mathbin{\hat{*}} r & \triangleq & (p \mathbin{\hat{*}} r) \vee (q \mathbin{\hat{*}} r) \\
(p \vee q) \mathbin{\hat{*}} r & \triangleq & (p \mathbin{\hat{*}} r) \vee (q \mathbin{\hat{*}} r) \\
\exists x.\, p \mathbin{\hat{*}} p & \triangleq & \exists x.\, (p \mathbin{\hat{*}} r) \\
(p \implies q) \mathbin{\hat{*}} r & \triangleq & (p \mathbin{\hat{*}} r) \implies (q \mathbin{\hat{*}} r) \\
\vdash p, p' \texttt{ agree} & \iff & \exists r, r', r'' \neq \texttt{False}.\, p * r \mathbin{\hat{*}} p' * r' = r'' \\
R \vdash p, q \texttt{ merge } q' & \iff & \forall p_R \leadsto q_R \in R. \vdash p, p_R \texttt{ agree} \wedge \vdash q, q_R \texttt{ agree} \\
 & & \implies (\exists r, r', r_R.\, (q * r) \mathbin{\hat{*}} (q_R * r_R) \implies q' * r' \wedge q \implies q') \\
R \vdash q \texttt{ stable} & \iff & \forall p_R \leadsto q_R \in R.\, \exists p.\, q \overset{\texttt{retag}\ r}{\Longrightarrow} p \wedge \vdash p, p_R \texttt{ agree} \wedge \\
 & & \exists r, r_R.\, (q * r) \mathbin{\hat{*}} (q_R * r_R) = q * r
\end{array}
$$

$$
\frac{\begin{array}{c} P \overset{\texttt{tag}\ r}{\Longrightarrow} p \quad \vdash \{p\}\ \mathbb{C}\ \{q\} \quad (p \leadsto q) \in G^* \\ R \vdash p, q \texttt{ merge } q' \quad R \vdash q' \texttt{ stable} \quad q' \overset{\texttt{notag}}{\Longrightarrow} Q \end{array}}{R, G \vdash \{P\}\ [\mathbb{C}]\ \{Q\}}\ \textit{commit}
$$

$$\frac{R, G \vdash \{P \wedge \mathbb{B}\}\ \mathbb{P}_1\ \{Q\} \qquad R, G \vdash \{P \wedge \neg\mathbb{B}\}\ \mathbb{P}_2\ \{Q\}}{R, G \vdash \{P\}\ \texttt{if}\,(\mathbb{B})\ \mathbb{P}_1\ \texttt{else}\ \mathbb{P}_2\ \{Q\}}\ \textit{choice}$$

$$\frac{R, G \vdash \{P\}\ \mathbb{P}_1\ \{Q'\} \qquad Q' \implies P' \quad R, G \vdash \{P'\}\ \mathbb{P}_2\ \{Q\}}{R, G \vdash \{P\}\ \mathbb{P}_1\,;\mathbb{P}_2\ \{Q\}}\ \textit{seq}$$

$$\frac{\begin{array}{l} R \cup G_2, G_1 \vdash \{P_1\}\ \mathbb{P}_1\ \{Q_1\} \qquad R \cup G_1, G_2 \vdash \{P_2\}\ \mathbb{P}_2\ \{Q_2\} \\ R \cup G_2 \vdash P_1\ \texttt{stable} \qquad R \cup G_1 \vdash P_2\ \texttt{stable} \end{array}}{R, G_1 \cup G_2 \vdash \{P_1 * P_2\}\ \mathbb{P}_1 \parallel \mathbb{P}_2\ \{Q_1 * Q_2\}}\ \textit{concur}$$

$$\frac{R, G \vdash \{P \wedge \mathbb{B}\}\ \mathbb{P}\ \{P\}}{R, G \vdash \{P\}\ \texttt{while}\,(\mathbb{B})\ \mathbb{P}\ \{P \wedge \neg\mathbb{B}\}}\ \textit{repeat}$$

4 blablabla

We use $l[-]$ to denote a location in either global or local heap.

$$\{l_x \mapsto 0 * l_y \mapsto 0\}$$

$$\begin{array}{l||l}
R : l_x \mapsto_r \_ * l_y \mapsto_r 0 \leadsto l_x \mapsto_w 1 * l_y \mapsto_r 0 & \\
\{l_x \mapsto 0 * l_y \mapsto 0 \vee l_x \mapsto 1 * l_y \mapsto 0\} & R : l_x \mapsto_r 0 * l_y \mapsto_r \_ \leadsto l_x \mapsto_r 0 * l_y \mapsto_w 1 \\
\left[\begin{array}{l}
\left\{\begin{array}{l} l_x \mapsto_r 0 * l_y \mapsto_r 0 \vee \\ l_x \mapsto_r 1 * l_y \mapsto_r 0 \end{array}\right\} \\
\mathtt{x} := [l_x]; \\
\left\{\begin{array}{l} l_x \mapsto_r 0 * l_y \mapsto_r 0 \wedge \mathtt{x} = 0 \vee \\ l_x \mapsto_r 1 * l_y \mapsto_r 0 \wedge \mathtt{x} = 1 \end{array}\right\} \\
\texttt{if}\,(\mathtt{x} = 0) \\
\quad [l_y] := 1; \\
\left\{\begin{array}{l} l_x \mapsto_r 0 * l_y \mapsto_w 1 \wedge \mathtt{x} = 0 \vee \\ l_x \mapsto_r 1 * l_y \mapsto_r 0 \wedge \mathtt{x} = 1 \end{array}\right\} \\
\mathit{MERGE} \\
\left\{\begin{array}{l} l_x \mapsto_r 0 * l_y \mapsto_w 1 \vee \\ l_x \mapsto_r 1 * l_y \mapsto_r 0 \vee \\ l_x \mapsto_w 1 * l_y \mapsto_w 1 \vee \\ l_x \mapsto_w 1 * l_y \mapsto_r 0 \end{array}\right\}
\end{array}\right]
&
\begin{array}{l}
\{l_x \mapsto 0 * l_y \mapsto 0 \vee l_x \mapsto 0 * l_y \mapsto 1\} \\
\left[\begin{array}{l}
\left\{\begin{array}{l} l_x \mapsto_r 0 * l_y \mapsto_r 0 \vee \\ l_x \mapsto_r 0 * l_y \mapsto_r 1 \end{array}\right\} \\
\mathtt{y} := [l_y]; \\
\texttt{if}\,(\mathtt{y} = 0) \\
\quad [l_x] := 1; \\
\left\{\begin{array}{l} l_x \mapsto_w 1 * l_y \mapsto_r 0 \vee \\ l_x \mapsto_r 0 * l_y \mapsto_r 1 \end{array}\right\} \\
\mathit{MERGE} \\
\left\{\begin{array}{l} l_x \mapsto_w 1 * l_y \mapsto_r 0 \vee \\ l_x \mapsto_r 0 * l_y \mapsto_r 1 \vee \\ l_x \mapsto_w 1 * l_y \mapsto_w 1 \vee \\ l_x \mapsto_r 0 * l_y \mapsto_w 1 \end{array}\right\}
\end{array}\right] \\
\left\{\begin{array}{l} l_x \mapsto 0 * l_y \mapsto 1 \vee \\ l_x \mapsto 1 * l_y \mapsto 0 \vee \\ l_x \mapsto 1 * l_y \mapsto 1 \end{array}\right\}
\end{array} \\
\left\{\begin{array}{l} l_x \mapsto 0 * l_y \mapsto 1 \vee \\ l_x \mapsto 1 * l_y \mapsto 0 \vee \\ l_x \mapsto 1 * l_y \mapsto 1 \end{array}\right\} &
\end{array}$$

$$\{l_x \mapsto 0 * l_y \mapsto 1 \vee l_x \mapsto 1 * l_y \mapsto 0 \vee l_x \mapsto 1 * l_y \mapsto 1\}$$

***The second rely might be useful in term of killing the possible states of other thread, even though it is irrelevant for the current thread. About the problem, an ugly solution is: if the resource appear in rely, current thread should keep it whether uses it or not. However this solution is against the idea of separation logic, i.e. we do not need to take care of those resource untouched.

$$l_y \mapsto_r \_ \rightsquigarrow l_y \mapsto_w 1$$
$$R : l_x \mapsto_r x * l_y \mapsto_r y * l_{tx2} \mapsto_r \_ * l_{ty2} \mapsto_r \_$$
$$\rightsquigarrow l_x \mapsto_r x * l_y \mapsto_r y * l_{tx2} \mapsto_w x * l_{ty2} \mapsto_w y$$
$$\left\{\begin{array}{l} l_x \mapsto 0 * l_y \mapsto 0 * l_{tx1} \mapsto 0 * l_{ty1} \mapsto 0 \vee \\ l_x \mapsto 0 * l_y \mapsto 1 * l_{tx1} \mapsto 0 * l_{ty1} \mapsto 0 \end{array}\right\}$$
$$[[l_x] := 1;]$$
$$\left\{\begin{array}{l} l_x \mapsto 1 * l_y \mapsto 0 * l_{tx1} \mapsto 0 * l_{ty1} \mapsto 0 \vee \\ l_x \mapsto 1 * l_y \mapsto 1 * l_{tx1} \mapsto 0 * l_{ty1} \mapsto 0 \end{array}\right\}$$
$$\left[\begin{array}{l} \left\{\begin{array}{l} l_x \mapsto_r 1 * l_y \mapsto_r 0 * l_{tx1} \mapsto_r 0 * l_{ty1} \mapsto_r 0 \vee \\ l_x \mapsto_r 1 * l_y \mapsto_r 1 * l_{tx1} \mapsto_r 0 * l_{ty1} \mapsto_r 0 \end{array}\right\} \\ \mathtt{x} := [l_x]; \\ [l_{tx1}] := \mathtt{x}; \\ \mathtt{y} := [l_y]; \\ [l_{ty1}] := \mathtt{y}; \\ \left\{\begin{array}{l} l_x \mapsto_r 1 * l_y \mapsto_r 0 * l_{tx1} \mapsto_w 1 * l_{ty1} \mapsto_w 0 \vee \\ l_x \mapsto_r 1 * l_y \mapsto_r 1 * l_{tx1} \mapsto_w 1 * l_{ty1} \mapsto_w 1 \end{array}\right\} \\ MERGE \\ \left\{\begin{array}{l} l_x \mapsto_r 1 * l_y \mapsto_r 0 * l_{tx1} \mapsto_w 1 * l_{ty1} \mapsto_w 0 \vee \\ l_x \mapsto_r 1 * l_y \mapsto_r 1 * l_{tx1} \mapsto_w 1 * l_{ty1} \mapsto_w 1 \vee \\ l_x \mapsto_r 1 * l_y \mapsto_w 1 * l_{tx1} \mapsto_w 1 * l_{ty1} \mapsto_w 0 \end{array}\right\} \end{array}\right]$$
$$\left\{\begin{array}{l} l_x \mapsto 1 * l_y \mapsto 0 * l_{tx1} \mapsto 1 * l_{ty1} \mapsto 0 \vee \\ l_x \mapsto 1 * l_y \mapsto 1 * l_{tx1} \mapsto 1 * l_{ty1} \mapsto 1 \vee \\ l_x \mapsto 1 * l_y \mapsto 1 * l_{tx1} \mapsto 1 * l_{ty1} \mapsto 0 \end{array}\right\}$$

$$l_x \mapsto_r \_ \rightsquigarrow l_x \mapsto_w 1$$
$$R : l_x \mapsto_r x * l_y \mapsto_r y * l_{tx1} \mapsto_r \_ * l_{ty1} \mapsto_r \_$$
$$\rightsquigarrow l_x \mapsto_r x * l_y \mapsto_r y * l_{tx1} \mapsto_w x * l_{ty1} \mapsto_w y$$
$$\left\{\begin{array}{l} l_x \mapsto 0 * l_y \mapsto 0 * l_{tx2} \mapsto 0 * l_{ty2} \mapsto 0 \vee \\ l_x \mapsto 1 * l_y \mapsto 0 * l_{tx2} \mapsto 0 * l_{ty2} \mapsto 0 \end{array}\right\}$$
$$[[l_y] := 1;]$$
$$\left\{\begin{array}{l} l_x \mapsto 0 * l_y \mapsto 1 * l_{tx2} \mapsto 0 * l_{ty2} \mapsto 0 \vee \\ l_x \mapsto 1 * l_y \mapsto 1 * l_{tx2} \mapsto 0 * l_{ty2} \mapsto 0 \end{array}\right\}$$
$$\left[\begin{array}{l} \mathtt{x} := [l_x]; \\ [l_{tx2}] := \mathtt{x}; \\ \mathtt{y} := [l_y]; \\ [l_{ty2}] := \mathtt{y}; \\ MERGE \\ \left\{\begin{array}{l} l_x \mapsto_r 0 * l_y \mapsto_r 1 * l_{tx2} \mapsto_w 0 * l_{ty2} \mapsto_w 1 \vee \\ l_x \mapsto_r 1 * l_y \mapsto_r 1 * l_{tx2} \mapsto_w 1 * l_{ty2} \mapsto_w 1 \vee \\ l_x \mapsto_w 1 * l_y \mapsto_r 1 * l_{tx2} \mapsto_w 0 * l_{ty2} \mapsto_w 1 \end{array}\right\} \end{array}\right]$$
$$\left\{\begin{array}{l} l_x \mapsto 0 * l_y \mapsto 1 * l_{tx2} \mapsto 0 * l_{ty2} \mapsto 1 \vee \\ l_x \mapsto 1 * l_y \mapsto 1 * l_{tx2} \mapsto 1 * l_{ty2} \mapsto 1 \vee \\ l_x \mapsto 1 * l_y \mapsto 1 * l_{tx2} \mapsto 0 * l_{ty2} \mapsto 1 \end{array}\right\}$$

$$\left\{\begin{array}{l} l_x \mapsto 1 * l_y \mapsto 1 * \\ \left(\begin{array}{l} l_{tx1} \mapsto 1 * l_{ty1} \mapsto 0 * l_{tx2} \mapsto 0 * l_{ty2} \mapsto 1 \vee \\ l_{tx1} \mapsto 1 * l_{ty1} \mapsto 0 * l_{tx2} \mapsto 1 * l_{ty2} \mapsto 1 \vee \\ l_{tx1} \mapsto 1 * l_{ty1} \mapsto 1 * l_{tx2} \mapsto 0 * l_{ty2} \mapsto 1 \vee \\ l_{tx1} \mapsto 1 * l_{ty1} \mapsto 1 * l_{tx2} \mapsto 1 * l_{ty2} \mapsto 1 \end{array}\right) \end{array}\right\}$$

Above, the 1 0 0 1 case will never happen. This is called long fork in snapshot isolation, for a single machine snapshot isolation, it should not happen.

We use $(-,-,-,-,-,-)$ to refer x,y,tx1,ty1,tx2,ty2.

$$l_y \mapsto_r \_ \rightsquigarrow l_y \mapsto_w 1$$
$$R : l_x \mapsto_r x * l_y \mapsto_r y * l_{tx2} \mapsto_r \_ * l_{ty2} \mapsto_r \_$$
$$\rightsquigarrow l_x \mapsto_r x * l_y \mapsto_r y * l_{tx2} \mapsto_w x * l_{ty2} \mapsto_w y$$
$$\{(0,0,0,0,0,0) \vee (0,1,0,0,0,0) \vee (0,1,0,0,0,1)\}$$
$$\left[\begin{array}{l} \{(0,0,0,0,0,0) \vee (0,1,0,0,0,0) \vee (0,1,0,0,0,1)\} \\ [l_x] := 1; \\ \{(1,0,0,0,0,0) \vee (1,1,0,0,0,0) \vee (1,1,0,0,0,1)\} \\ MERGE \\ \{(1,0,0,0,0,0) \vee (1,1,0,0,0,0) \vee (1,1,0,0,0,1)\} \end{array}\right]$$
$$\left\{\begin{array}{l} (1,0,0,0,0,0) \vee (1,1,0,0,0,0) \vee (1,1,0,0,0,1) \vee \\ (1,0,0,0,1,0) \vee (1,1,0,0,1,1) \end{array}\right\}$$
$$\left[\begin{array}{l} \left\{\begin{array}{l} (1,0,0,0,0,0) \vee (1,1,0,0,0,0) \vee (1,1,0,0,0,1) \vee \\ (1,0,0,0,1,0) \vee (1,1,0,0,1,1) \end{array}\right\} \\ \mathtt{x} := [l_x]; \\ [l_{tx1}] := \mathtt{x}; \\ \mathtt{y} := [l_y]; \\ [l_{ty1}] := \mathtt{y}; \\ \left\{\begin{array}{l} (1,0,1,0,0,0) \vee (1,1,1,1,0,0) \vee (1,1,1,1,0,1) \vee \\ (1,0,1,0,1,0) \vee (1,1,1,1,1,1) \end{array}\right\} \\ MERGE \\ \left\{\begin{array}{l} (1,0,1,0,0,0) \vee (1,1,1,1,0,0) \vee (1,1,1,1,0,1) \vee \\ (1,0,1,0,1,0) \vee (1,1,1,1,1,1) \vee (1,1,1,0,0,0) \end{array}\right\} \end{array}\right]$$
$$\left\{\begin{array}{l} (1,0,1,0,0,0) \vee (1,1,1,1,0,0) \vee (1,1,1,1,0,1) \vee \\ (1,0,1,0,1,0) \vee (1,1,1,1,1,1) \vee (1,1,1,0,0,0) \vee \\ (1,1,1,0,1,1) \end{array}\right\}$$

$$l_x \mapsto_r \_ \rightsquigarrow l_x \mapsto_w 1$$
$$R : l_x \mapsto_r x * l_y \mapsto_r y * l_{tx1} \mapsto_r \_ * l_{ty1} \mapsto_r \_$$
$$\rightsquigarrow l_x \mapsto_r x * l_y \mapsto_r y * l_{tx1} \mapsto_w x * l_{ty1} \mapsto_w y$$
$$\{(0,0,0,0,0,0) \vee (1,0,0,0,0,0) \vee (1,0,1,0,0,0)\}$$
$$\left[\begin{array}{l} \{(0,0,0,0,0,0) \vee (1,0,0,0,0,0) \vee (1,0,1,0,0,0)\} \\ [l_y] := 1; \\ \{(0,1,0,0,0,0) \vee (1,1,0,0,0,0) \vee (1,1,1,0,0,0)\} \end{array}\right]$$
$$\left\{\begin{array}{l} (0,1,0,0,0,0) \vee (1,1,0,0,0,0) \vee (1,1,1,0,0,0) \vee \\ (0,1,0,1,0,0) \vee (1,1,1,1,0,0) \end{array}\right\}$$
$$\left[\begin{array}{l} \left\{\begin{array}{l} (0,1,0,0,0,0) \vee (1,1,0,0,0,0) \vee (1,1,1,0,0,0) \vee \\ (0,1,0,1,0,0) \vee (1,1,1,1,0,0) \end{array}\right\} \\ \mathtt{x} := [l_x]; \\ [l_{tx2}] := \mathtt{x}; \\ \mathtt{y} := [l_y]; \\ [l_{ty2}] := \mathtt{y}; \\ \left\{\begin{array}{l} (0,1,0,0,0,1) \vee (1,1,0,0,1,1) \vee (1,1,1,0,1,1) \vee \\ (0,1,0,1,0,1) \vee (1,1,1,1,1,1) \end{array}\right\} \\ MERGE \\ \left\{\begin{array}{l} (0,1,0,0,0,1) \vee (1,1,0,0,1,1) \vee (1,1,1,0,1,1) \vee \\ (0,1,0,1,0,1) \vee (1,1,1,1,1,1) \vee (1,1,0,0,0,1) \end{array}\right\} \end{array}\right]$$
$$\left\{\begin{array}{l} (0,1,0,0,0,1) \vee (1,1,0,0,1,1) \vee (1,1,1,0,1,1) \vee \\ (0,1,0,1,0,1) \vee (1,1,1,1,1,1) \vee (1,1,0,0,0,1) \vee \\ (1,1,1,1,0,1) \end{array}\right\}$$

$$\{(1,1,1,0,1,1) \vee (1,1,1,1,1,1) \vee (1,1,1,1,0,1)\}$$

$$\begin{bmatrix}[l_x] := 1; \\ [l_z] := 1;\end{bmatrix} \parallel \begin{bmatrix}[l_y] := 2; \\ [l_z] := 2;\end{bmatrix}$$