1 Litmus Tests

$$\left[\begin{array}{l}\mathtt{x} := [a_x];\\ \mathtt{if}\,(\mathtt{x} = 0)\\ \quad [a_y] := 1\end{array}\right] \;\Bigg\|\; \left[\begin{array}{l}\mathtt{y} := [a_y];\\ \mathtt{if}\,(\mathtt{x} = 0)\\ \quad [a_x] := 1\end{array}\right] \;\Bigg\|\; \big[[a_x] := 2\big] \;\Bigg\|\; \big[[a_y] := 2\big]$$

$$\left[\begin{array}{l}\mathtt{x} := [a_x];\\ \mathtt{if}\,(\mathtt{x} = 0)\\ \quad [a_y] := 1\end{array}\right] \;\Bigg\|\; \left[\begin{array}{l}\mathtt{y} := [a_y];\\ \mathtt{if}\,(\mathtt{x} = 0)\\ \quad [a_x] := 1\end{array}\right]$$

$$\begin{array}{c}\big[[a_x] := 1;\big]\\ \left[\begin{array}{l}\mathtt{x} := [a_x];\\ \mathtt{y} := [a_y];\\ \mathtt{if}\,(\mathtt{x} = 1 \;\mathtt{and}\; \mathtt{y} = 0)\\ \quad [a_w] := 1\end{array}\right]\end{array} \;\Bigg\|\; \begin{array}{c}\big[[a_y] := 1;\big]\\ \left[\begin{array}{l}\mathtt{x} := [a_x];\\ \mathtt{y} := [a_y];\\ \mathtt{if}\,(\mathtt{x} = 0 \;\mathtt{and}\; \mathtt{y} = 1)\\ \quad [a_z] := 1\end{array}\right]\end{array}$$

2 semantics

Definition 1 (Program values)**.** Assume a countably infinite set of *addresses*, $a \in \textsc{Addr}$, and a countably infinite set of *program variables* $\mathtt{x} \in \textsc{Var}$. The set of *program values* is $v \in \textsc{Val} \triangleq \mathbb{N} \cup \textsc{Addr}$, where $\mathbb{N}$ denotes the set of natural numbers.

Definition 2 (Stacks)**.** Given the set of program variables $\textsc{Var}$ and the set of program values $\textsc{Val}$ (Def. 1), the set of variable stacks is $\sigma \in \textsc{Stack} \triangleq \textsc{Var} \overset{fin}{\rightharpoonup} \textsc{Val}$.

Our programs (ranged over by $\mathbb{P}$) are defined by an inductive grammar comprising the standard constructs of `skip`, sequential composition ($\mathbb{P};\mathbb{P}$), non-deterministic choice ($\mathbb{P}+\mathbb{P}$), loops ($\mathbb{P}^*$) and parallel composition ($\mathbb{P}\|\mathbb{P}$). Additionally, our programming language contains the *transaction* construct $[\mathbb{T}]$ denoting the *atomic* execution of the transaction $\mathbb{T}$. The atomicity guarantees of this execution are dictated by the underlying consistency model (snapshot isolation in this case). Transactions (ranged over by $\mathbb{T}$) are defined by a similar inductive grammar comprising `skip`, non-deterministic choice, loops and sequential composition, as well as constructs for assignment, lookup and update. Transactions do *not* contain the *parallel* composition construct ($\|$) as they are to be executed atomically.

Definition 3 (Programming language)**.** The set of *programs*, $\mathbb{P} \in \textsc{Prog}$, is defined by the following grammar:

$$\mathbb{P} ::= \mathtt{skip} \ \mid \ [\mathbb{T}] \ \mid \ \mathbb{P};\mathbb{P} \ \mid \ \mathbb{P}+\mathbb{P} \ \mid \ \mathbb{P}^* \ \mid \ \mathbb{P}\,\|\,\mathbb{P}$$

The $\mathbb{T} \in \textsc{Trans}$ in the grammar above denotes a *transaction* defined by the following grammar:

$$\mathbb{T} ::= \mathtt{skip} \ \mid \ \mathtt{x} := \mathbb{E} \ \mid \ [\mathbb{E}] := \mathbb{E} \ \mid \ \mathtt{x} := [\mathbb{E}] \ \mid \ \mathbb{T}+\mathbb{T} \ \mid \ \mathbb{T}^* \ \mid \ \mathbb{T};\mathbb{T}$$

where $\mathbb{E} \in \textsc{Expr}$ denotes an *arithmetic expression* defined by the grammar below with $v \in \textsc{Val}$ and $\mathtt{x} \in \textsc{Var}$ (Def. 1).

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

Given a stack $\sigma \in \textsc{Stack}$ (Def. 2), the *expression evaluation* function, $[\![.]\!]_{(.)} : \textsc{Expr} \times \textsc{Stack} \rightharpoonup \textsc{Val}$, is defined inductively over the structure of expressions as follows:

$$\begin{array}{rcl} [\![v]\!]_\sigma & \triangleq & v \\ [\![\mathtt{x}]\!]_\sigma & \triangleq & \sigma(v) \\ [\![\mathbb{E}_1 + \mathbb{E}_2]\!]_\sigma & \triangleq & [\![\mathbb{E}_1]\!]_\sigma + [\![\mathbb{E}_2]\!]_\sigma \\ [\![\mathbb{E}_1 * \mathbb{E}_2]\!]_\sigma & \triangleq & [\![\mathbb{E}_1]\!]_\sigma * [\![\mathbb{E}_2]\!]_\sigma \end{array}$$

We model the global database state as a *timestamp heap*. A timestamp heap is a partial function from addresses to *histories*. A history is a partial function from timestamps to a set of *events*. An event is a triple comprising the value read or written, the identifier of the transaction carrying out the event, and an *event tag* denoting a *start event* (`S`), an *end* event (`E`), a *read* event (`R`) or a *write* event (`W`). We model our timestamps, $t \in \textsc{TimeStamp}$, as elements of an (uncountably) infinite set with a total order relation $<$. To ensure the availability of an appropriate timestamp, we assume that the timestamp set $\textsc{TimeStamp}$ is *dense*. That is, given any two timestamps t_1 and t_2 such that $t_1 < t_2$, an intermediate timestamp t can be found such that $t_1 < t < t_2$.

Definition 4 (Timestamp heaps)**.** Assume a set of *timestamps*, $t \in \textsc{TimeStamp}$. Assume a *total order* relation on timestamps, $< : \textsc{TimeStamp} \times \textsc{TimeStamp}$, such that:

$$\forall t_1, t_2 \in \textsc{TimeStamp}.\ t_1 < t_2 \vee t_2 < t_1$$

Assume that the timestamp set $\textsc{TimeStamp}$ is *dense* with respect to the ordering relation $<$. That is,

$$\forall t_1, t_2.\ t_1 < t_2 \Rightarrow \exists t.\ t_1 < t < t_2$$

The set of *event tags* is: $e \in \textsc{ETag} \triangleq \{\mathtt{S}, \mathtt{E}, \mathtt{R}, \mathtt{W}\}$.
Assume a countably infinite set of *transaction identifiers* $\alpha, \beta \in \textsc{TransID}$. Given the set of program values $\textsc{Val}$ (Def. 1), the set of *timestamp heaps* is defined as $H \in \textsc{TSHeap} \triangleq \textsc{Addr} \overset{fin}{\rightharpoonup} (\textsc{Val} \times \textsc{TransID} \times \textsc{ETag})$.
The *timestamp heap composition function*, $\bullet_H : \textsc{TSHeap} \times \textsc{TSHeap} \rightharpoonup \textsc{TSHeap}$, is defined as follows, for all $a \in \textsc{Addr}$, where $\uplus$ denotes the standard disjoint function union:

$$(H_1 \bullet_H H_2)(a) \triangleq \begin{cases} H_1(a) \uplus H_2(a) & \text{if } a \in \mathrm{dom}(H_1) \text{ and } a \in \mathrm{dom}(H_2) \\ H_1(r) & \text{if } a \in \mathrm{dom}(H_1) \text{ and } a \notin \mathrm{dom}(H_2) \\ H_2(r) & \text{if } a \notin \mathrm{dom}(H_1) \text{ and } a \in \mathrm{dom}(H_2) \\ \text{undefined} & \text{otherwise} \end{cases}$$

The *timestamp heap unit element* is $\mathbf{0}_H \triangleq \emptyset$, denoting a function with an empty domain. The *partial commutative monoid of timestamp heaps* is $(\textsc{TSHeap}, \bullet_h, \{\mathbf{0}_H\})$.

We write $t_1 \leq t_2$ as a shorthand for $t_1 < t_2 \vee t_1 = t_2$. We write $t_1 \oplus t \odot t_2$ for $t_1 \oplus t \wedge t \odot t_2$, where $\oplus, \odot \in \{<, \leq\}$. We often use the mirrored symbols and write $t_1 > t_2$ (resp. $t_1 \geq t_2$) for $t_2 < t_1$ (resp. $t_2 \leq t_1$). Lastly, we use the standard interval notation and write:

$$\begin{array}{lll} (t_1, t_2) & \text{for} & \{t \mid t_1 < t < t_2\} \\ (t_1, t_2] & \text{for} & \{t \mid t_1 < t \leq t_2\} \\ [t_1, t_2) & \text{for} & \{t \mid t_1 \leq t < t_2\} \\ [t_1, t_2] & \text{for} & \{t \mid t_1 \leq t \leq t_2\} \end{array}$$

Whilst the state of the database is given by globally-accessed (by all threads) timestamp heaps, we use *fingerprint heaps* to model a *local snapshot* of the global timestamp heaps made available to transactions during their execution. In order to successfully commit a transaction and avoid conflicts, we must track the *fingerprint* of a transaction, namely those addresses read from or written to by the transaction. As such, we model fingerprint heaps as finite partial maps from addresses to pairs comprising a value and a fingerprint. The fingerprint of an address may be 1) $\emptyset$ denoting that the address has not been touched; 2) $\{\mathtt{r}\}$ denoting that the address has been read; 3) $\{\mathtt{w}\}$ denoting that the address has been mutated (written to); and 4) $\{\mathtt{r}, \mathtt{w}\}$ denoting that the address has been initially read and subsequently mutated.

Definition 5 (Fingerprint heaps)**.** The set of *fingerprints* is $f \in \textsc{FingerPrint} \triangleq \mathcal{P}(\{\mathtt{w}, \mathtt{r}\})$. Given the sets of program values $\textsc{Val}$ (Def. 1) and addresses $\textsc{Addr}$ (Def. 4), the set of *fingerprint heaps* is: $h \in \text{FPH}\textsc{eap} \triangleq \textsc{Addr} \overset{fin}{\rightharpoonup} (\textsc{Val} \times \textsc{FingerPrint})$. The *fingerprint heap composition function*, $\bullet_h : \text{FPH}\textsc{eap} \times \text{FPH}\textsc{eap} \rightharpoonup \text{FPH}\textsc{eap}$, is defined as $\bullet_h \triangleq \uplus$, where $\uplus$ denotes the standard disjoint function union. The *fingerprint heap unit element* is $\mathbf{0}_h \triangleq \emptyset$, denoting a function with an empty domain. The *partial commutative monoid of fingerprint heaps* is $(\text{FPH}\textsc{eap}, \bullet_h, \{\mathbf{0}_h\})$.

Given a fingerprint heap h and an address a, we write $h^{\mathsf{v}}(a)$ and $h^{\mathsf{fp}}(a)$ for the first and second projections of $h(a)$, respectively. We write $\mathbf{0}_h$ for a fingerprint heap with an empty domain. We write $h_1 \uplus h_2$ for the standard disjoint function union of h_1 and h_2.

We introduce two update functions on fingerprints, $f \overset{\mathtt{r}}{\hookleftarrow}$ and $f \overset{\mathtt{w}}{\hookleftarrow}$, for updating a fingerprint f. Intuitively, the $f \overset{\mathtt{w}}{\hookleftarrow}$ always *extends* f with the $\mathtt{w}$ fingerprint. On the other hand, the $f \overset{\mathtt{r}}{\hookleftarrow}$ extends f with $\mathtt{r}$ *only if* f does not already contain the $\mathtt{w}$ fingerprint (i.e. $\mathtt{w} \notin f$). This is to capture the fact that once an address is written to and thus the fingerprint contains $\mathtt{w}$, the following reads from the same address are considered local and need not be recorded in the fingerprint.

Definition 6 (Fingerprint extension)**.** Given the set of fingerprints $\textsc{FingerPrint}$ (Def. 5), the *write fingerprint extension* function, $(.) \overset{\mathtt{w}}{\hookleftarrow} . : \textsc{FingerPrint} \to \textsc{FingerPrint}$, and the *write fingerprint extension* function, $(.) \overset{\mathtt{w}}{\hookleftarrow} . : \textsc{FingerPrint} \to \textsc{FingerPrint}$, are defined as follows, for all $f \in \textsc{FingerPrint}$:

$$f \overset{\mathtt{w}}{\hookleftarrow} \triangleq f \cup \{\mathtt{w}\}$$
$$f \overset{\mathtt{r}}{\hookleftarrow} \triangleq \begin{cases} f \cup \{\mathtt{r}\} & \text{if } \mathtt{w} \notin f \\ f & \text{otherwise} \end{cases}$$

We define the operational semantics of transactions $\mathbb{T}$ with respect to a pair of the form (σ, h) comprising a (local) stack and a (local) fingerprint heap corresponding to a snapshot of the global timestamp heap. The operational semantics of transactions is given in Fig. 1. The operational semantics of transactions is standard with the exception of the TRead and TWrite rules where the fingerprint of the address read from (resp. written to) is extended with $\mathtt{r}$ (resp. $\mathtt{w}$).

Definition 7 (Transaction semantics)**.** Given the sets of stacks (Def. 2), fingerprint heaps (Def. 5) and transactions (Def. 3), the *operational semantics of transactions*, $\rightsquigarrow_l$: $((\textsc{Stack} \times \text{FPH}\textsc{eap}) \times \textsc{Trans}) \times ((\textsc{Stack} \times \text{FPH}\textsc{eap}) \times \textsc{Trans})$, is given in Fig. 1.

In order to formulate the operational semantics of a program $\mathbb{P}$, we extend the programming language with an auxiliary wait construct, $\mathtt{wait}(i)$, added to programs as a suffix to denote thread joining points. Intuitively, the $\mathtt{wait}(i)$ construct indicates that the current thread is waiting on the thread identified by i to finish its execution and join the current thread. We refer to the programs produced by this extended syntax as *intermediate programs*. This is because the $\mathtt{wait}(.)$ construct yields additional programs that cannot be written by the clients of the database and is merely used to capture intermediate steps during parallel execution.

We define the per-thread operational semantics of programs with respect to a triple of the form (σ, H, t) comprising a (locally-accessed) stack, a (globally-accessed) timestamp heap, and a (locally-recorded) timestamp. Each step of the per-thread operational semantics is decorated with a *label* recording the action taken by the thread. In particular, a label may be $\mathtt{id}$, denoting an identity transition; $\mathtt{cmt}(\alpha)$, denoting the committing of

$$\frac{[\![\mathbb{E}]\!]_\sigma = v}{(\sigma, h), \mathtt{x} := \mathbb{E} \ \rightsquigarrow_l \ (\sigma[\mathtt{x} \mapsto v], h), \mathtt{skip}} \ \text{TAss}$$

$$\frac{[\![\mathbb{E}_1]\!]_\sigma = a \qquad [\![\mathbb{E}_2]\!]_\sigma = v \qquad h(l){=}(-, f)}{(\sigma, h), [\mathbb{E}_1] := \mathbb{E}_2 \ \rightsquigarrow_l \ (\sigma, h[a \mapsto (v, f \overset{\mathtt{w}}{\hookleftarrow})], \mathtt{skip}} \ \text{TWrite}$$

$$\frac{[\![\mathbb{E}]\!]_\sigma = a \qquad h(a){=}(v, f)}{(\sigma, h), \mathtt{x} := [\mathbb{E}] \ \rightsquigarrow_l \ (\sigma[\mathtt{x} \mapsto v], h[a \mapsto (v, f \overset{\mathtt{r}}{\hookleftarrow})], \mathtt{skip}} \ \text{TRead}$$

$$\frac{}{(\sigma, h), \mathbb{T}_1 + \mathbb{T}_2 \ \rightsquigarrow_l \ (\sigma, h), \mathbb{T}_1} \ \text{TChoiseL}$$

$$\frac{}{(\sigma, h), \mathbb{T}_1 + \mathbb{T}_2 \ \rightsquigarrow_l \ (\sigma, h), \mathbb{T}_2} \ \text{TChoiseR}$$

$$\frac{}{(\sigma, h), \mathbb{T}^* \ \rightsquigarrow_l \ (\sigma, h), \mathtt{skip} + (\mathbb{T}; \mathbb{T}^*)} \ \text{TLoop}$$

$$\frac{}{(\sigma, h), \mathtt{skip}; \mathbb{T}_2 \ \rightsquigarrow_l \ (\sigma, h), \mathbb{T}_2} \ \text{TSeqSkip}$$

$$\frac{(\sigma, h), \mathbb{T}_1 \ \rightsquigarrow_l \ (\sigma', h'), \mathbb{T}_1'}{(\sigma, h), \mathbb{T}_1; \mathbb{T}_2 \ \rightsquigarrow_l \ (\sigma', h'), \mathbb{T}_1'; \mathbb{T}_2} \ \text{TSeq}$$

Figure 1: The transaction operational semantics

transaction α; $\mathtt{fork}(i, \mathbb{P})$, denoting the forking a new thread i the execute program $\mathbb{P}$; or $\mathtt{join}(i, t)$, denoting the joining of thread i with the local timestamp t.

The per-thread operational semantics of programs is given in Fig. 2. With the exception of the PCommit, PPar and PWait, the remaining rules are straightforward.

Definition 8 (Thread semantics)**.** Assume a countably infinite set of thread identifiers $i, j \in \text{ThreadID}$. The set of *thread transition labels*, $\iota \in \text{Label}$, is defined by the following grammar, where $\mathbb{P}$ denotes a program (Def. 3), the α demotes a transaction identifier and t denotes a timestamp (Def. 4):

$$\iota \in \text{Label} ::= \mathtt{id} \mid \mathtt{cmt}(\alpha) \mid \mathtt{fork}(i, \mathbb{P}) \mid \mathtt{join}(i, t)$$

The set of *intermediate programs*, $\mathbb{P}^\uparrow \in \text{IProg}$, is defined by the following grammar:

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

Given the set of stacks Stack (Def. 2), timestamp heaps TSHeap and timestamps TimeStamp (Def. 4), The *per-thread operational semantics* of programs:

$$\rightsquigarrow_t : \big((\text{Stack} \times \text{TSHeap} \times \text{TimeStamp}) \times \text{IProg}\big) \ \times \text{Label} \ \times \big((\text{Stack} \times \text{TSHeap} \times \text{TimeStamp}) \times \text{IProg}\big)$$

is defined in Fig. 2.

The PCommit rule states that a transaction prophesies a starting timestamp t_s (at which point it takes a snapshot h_s and runs locally), and an ending timestamp t_e (at which point the transaction is successfully committed as ensured by the $\mathtt{can\_commit}$ predicate).

The Par rule forks a new thread and inserts the appropriate joining point by appending the auxiliary $\mathtt{wait}(i)$ operation, where i denotes the identifier of the newly forked thread. The Wait rule dually awaits the termination of thread i and subsequently updates its timestamp to the maximum value between its own timestamp and that of i. Note that these two rules are labelled with the $\mathtt{fork}(i, \mathbb{P})$ and $\mathtt{join}(i, t)$ which are used by the semantics of the thread pool described shortly.

In order to model concurrency, we use thread pools. A thread pool is modelled as a finite partial map from thread identifiers to triples of the form $(\sigma, t, \mathbb{P})$. That is, each thread is associated with a stack σ,a timestamp t and a program $\mathbb{P}$ to be executed.

$$\frac{\begin{array}{c} t \le t_s < t_e \qquad h_s = \mathsf{snapshot}(H, t_s) \qquad (\sigma, h_s), \mathbb{T} \rightsquigarrow_l^* (\sigma', h_e), \mathtt{skip} \\ \mathtt{can\_commit}(H, h_e, t_s, t_e) \quad \mathtt{fresh}(H, \alpha) \quad H' = \mathsf{commit}(H, h_s, h_e, \alpha, t_s, t_e) \end{array}}{(\sigma, H, t), \lceil \mathbb{T} \rceil \overset{\mathtt{cmt}(\alpha)}{\rightsquigarrow_t} (\sigma', H', t_e), \mathtt{skip}} \textsc{Commit}$$

$$\frac{}{(\sigma, H, t), \mathbb{P}_1 + \mathbb{P}_2 \overset{\mathtt{id}}{\rightsquigarrow_t} (\sigma, H, t), \mathbb{P}_1} \textsc{PChoiseL}$$

$$\frac{}{(\sigma, H, t), \mathbb{P}_1 + \mathbb{P}_2 \overset{\mathtt{id}}{\rightsquigarrow_t} (\sigma, H, t), \mathbb{P}_2} \textsc{PChoiseR}$$

$$\frac{}{(\sigma, H, t), \mathbb{P}^* \overset{\mathtt{id}}{\rightsquigarrow_t} (\sigma, H, t), \mathtt{skip} + (\mathbb{P}; \mathbb{P}^*)} \textsc{PLoop}$$

$$\frac{}{(\sigma, H, t), \mathtt{skip}; \mathbb{P}^\uparrow \overset{\mathtt{id}}{\rightsquigarrow_t} (\sigma, H, t), \mathbb{P}^\uparrow} \textsc{PSeqSkip}$$

$$\frac{(\sigma, H, t), \mathbb{P}_1^\uparrow \overset{\iota}{\rightsquigarrow_t} (\sigma', H', t'), \mathbb{P}_1^{\uparrow\prime}}{(\sigma, H, t), \mathbb{P}_1^\uparrow; \mathbb{P}_2^\uparrow \overset{\iota}{\rightsquigarrow_t} (\sigma', H', t'), \mathbb{P}_1^{\uparrow\prime}; \mathbb{P}_2^\uparrow} \textsc{PSeq}$$

$$\frac{}{(\sigma, H, t), \mathbb{P}_1 \parallel \mathbb{P}_2 \overset{\mathtt{fork}(i, \mathbb{P}_2)}{\rightsquigarrow_t} (\sigma, H, t), \mathbb{P}_1; \mathtt{wait}(i)} \textsc{PPar}$$

$$\frac{}{(\sigma, H, t), \mathtt{wait}(i) \overset{\mathtt{join}(i, t')}{\rightsquigarrow_t} (\sigma, H, \max\{t, t'\}), \mathtt{skip}} \textsc{PWait}$$

where

$$\begin{aligned}
\mathsf{snapshot}(.,.) &: \textsc{TSHeap} \times \textsc{TimeStamp} \rightharpoonup \textsc{FPHeap} \\
\mathsf{snapshot}(H, t) &\triangleq \lambda a. \begin{cases} (v, \emptyset) & \exists t' \le t.\ H(a)(t') {=} (v, \mathtt{W}, -) \wedge \forall t'' \in (t', t).\ H(a)(t'') {=} (-, \mathtt{R}, -) \\ \\ \text{undefined} & \text{otherwise} \end{cases} \\
\mathsf{commit}(., ., ., ., ., .) &: \textsc{TSHeap} \times \textsc{FPHeap} \times \textsc{FPHeap} \times \textsc{TransID} \times \textsc{TimeStamp} \times \textsc{TimeStamp} \rightarrow \textsc{TSHeap} \\
\mathsf{commit}(H, h_s, h_e, \alpha, t_s, t_e) &\triangleq \lambda a. \begin{cases} H(a)[t_s \mapsto (h_s^{\mathsf{v}}(a), \mathtt{S}, \alpha)][t_e \mapsto (h^{\mathsf{v}} h_e(a), \mathtt{W}, \alpha)] & \text{if } h_e^{\mathsf{fp}}(a) {=} \{\mathtt{w}\} \\ H(a)[t_s \mapsto (h_s^{\mathsf{v}}(a), \mathtt{R}, \alpha)][t_e \mapsto (h_e^{\mathsf{v}}(a), \mathtt{E}, \alpha)] & \text{if } h_e^{\mathsf{fp}}(a) {=} \{\mathtt{r}\} \\ H(a)[t_s \mapsto (h_s^{\mathsf{v}}(a), \mathtt{R}, \alpha)][t_e \mapsto (h_e^{\mathsf{v}}(a), \mathtt{W}, \alpha)] & \text{if } h_e^{\mathsf{fp}}(a) {=} \{\mathtt{r}, \mathtt{w}\} \\ H(a) & \text{otherwise} \end{cases} \\
\mathtt{can_commit}(H, h, t_s, t_e) &\triangleq \mathtt{wf_hist}(H, h, t_s, t_e) \wedge \mathtt{consistent}(H, h, t_s, t_e) \\
\mathtt{wf_hist}(H, h, t_s, t_e) &\triangleq \forall a.\ \forall t \in \{t_s, t_e\}.\ h^{\mathsf{fp}}(a) {\neq} \emptyset \Rightarrow H(a)(t) \uparrow \\
\mathtt{consistent}(H, h, t_s, t_e) &\triangleq \forall a.\ \mathtt{w} \in h^{\mathsf{fp}}(a) \Rightarrow \\
&\qquad \forall t \in (t_s, t_e).\ H(a)(t) {\neq} (-, \mathtt{W}, -) \\
&\qquad \wedge \neg \exists \alpha, t_s', t_e'.\ t_s' < t_e \wedge t_e' > t_e \wedge H(a)(t_s') = (-, -, \alpha) \wedge H(a)(t_e') = (-, \mathtt{W}, \alpha) \\
&\qquad \wedge \forall t_m.\ t_m {=} \min(\{t'' \mid t'' > t_e \wedge H(a)(t'') \downarrow\}) \Rightarrow H(a)(t_{min}) \neq (-, \mathtt{R}, -) \\
\mathtt{fresh}(H, \alpha) &\triangleq \neg \exists a, t.\ H(a)(t) {=} (-, -, \alpha)
\end{aligned}$$

Figure 2: Per-thread operational semantics

$$\frac{(\sigma, H, t), \mathbb{P}^{\uparrow} \overset{\iota}{\leadsto}_t (\sigma', H', t'), \mathbb{P}^{\uparrow\prime} \quad \iota \in \{\mathtt{id}, \mathtt{cmt}(-)\}}{(H, \eta \uplus \{i \mapsto (\sigma, t, \mathbb{P}^{\uparrow})\}) \overset{\iota}{\leadsto}_g (H', \eta \uplus \left\{i \mapsto (\sigma', t', \mathbb{P}^{\uparrow\prime})\right\})} \text{ PSINGLE}$$

$$\frac{(\sigma, H, t), \mathbb{P}^{\uparrow} \overset{\mathtt{fork}(i', \mathbb{P}'')}{\leadsto_t} (\sigma', H', t'), \mathbb{P}^{\uparrow\prime}}{(H, \eta \uplus \{i \mapsto (\sigma, t, \mathbb{P}^{\uparrow})\}) \overset{\mathtt{fork}(i', \mathbb{P}'')}{\leadsto_g} (H', \eta \uplus \left\{i \mapsto (\sigma', t', \mathbb{P}^{\uparrow\prime}), i' \mapsto (\lambda \mathtt{x}.\, 0, t', \mathbb{P}'')\right\})} \text{ PFORK}$$

$$\frac{(\sigma, H, t), \mathbb{P}^{\uparrow} \overset{\mathtt{join}(i', t'')}{\leadsto_t} (\sigma', H', t'), \mathbb{P}^{\uparrow\prime}}{(H, \eta \uplus \{i \mapsto (\sigma, t, \mathbb{P}^{\uparrow}), i' \mapsto (\sigma', t'', \mathtt{skip})\}) \overset{\mathtt{join}(i', t'')}{\leadsto_g} (H', \eta \uplus \left\{i \mapsto (\sigma', t', \mathbb{P}^{\uparrow\prime})\right\})} \text{ PJOIN}$$

Figure 3: Thread pool semantics

Definition 9 (Thread pools)**.** Given the sets of stacks STACK (Def. 2), timestamps TIMESTAMP (Def. 4) and programs PROG (Def. 3), the set of *thread pools* is: $\eta \in \text{THREADPOOL} \triangleq \text{THREADID} \overset{fin}{\rightharpoonup} \text{STACK} \times \text{TIMESTAMP} \times \text{PROG}$.

Definition 10 (Thread pool semantics)**.** Given the sets of timestamp heaps TSHEAP (Def. 4), transition labels (Def. 8) and thread pools THREADPOOL (Def. 9), the *thread pool semantics*,

$$\leadsto_g : (\text{TSHEAP} \times \text{THREADPOOL}) \times \text{LABEL} \times (\text{TSHEAP} \times \text{THREADPOOL})$$

is defined in Fig. 3.

The thread pool operational semantics is given in Fig. 3, where an arbitrary thread in the pool η is picked to run for one step. If the next execution step is a thread fork, then a new thread i is allocated in the pool to be executed with its stack and timestamp copied from those of the parent (forking) thread. Conversely, when the next execution step is the joining of thread i', then i' is removed from the thread pool and the local time stamp is accordingly updated to the maximum timestamp between the parent thread and that of i' (as guaranteed by the per-thread semantic relation $\leadsto_t$ in the premise).

3 Assertion and Rules

3.1 Local/Transaction

Definition 11 (capabilities)**.** Assume a partial commutative monoid for *primitive capabilities* $(\textsc{Kap}, \bullet_\kappa, \mathbf{0}_\kappa)$ with $\kappa \in \textsc{Kap}$. Assume a countably infinite set of region identifiers $r \in \textsc{RegionId}$. The set of *capabilities* is $c \in \textsc{Cap} \triangleq \textsc{RegionId} \rightharpoonup \textsc{Kap}$. The *capability composition function*, $\bullet_c : \textsc{Cap} \times \textsc{Cap} \rightharpoonup \textsc{Cap}$, is defined as follows:

$$(c_1 \bullet_c c_2)(r) \triangleq \begin{cases} c_1(r) \bullet_\kappa c_2(r) & \text{if } r \in \text{dom}(c_1) \text{ and } r \in \text{dom}(c_2) \\ c_1(r) & \text{if } r \in \text{dom}(c_1) \text{ and } r \notin \text{dom}(c_2) \\ c_2(r) & \text{if } r \notin \text{dom}(c_1) \text{ and } r \in \text{dom}(c_2) \\ \text{undefined} & \text{otherwise} \end{cases}$$

The *capability unit element*, $\mathbf{0}_c$, denotes a function with an empty domain. The *capability partial commutative monoid* is $(\textsc{Cap}, \bullet_c, \{\mathbf{0}_c\})$.

Definition 12 (Local state)**.** Given the set of fingerprint heaps FPHeap (Def. 5) and the set of capabilities Cap (Def. 11), the set of *local states* is $\ell \in \text{LState} \triangleq \text{FPHeap} \times \textsc{Cap}$. The *local state composition function*, $\bullet_\ell : \text{LState} \times \text{LState} \rightharpoonup \text{LState}$, is defined component-wise as: $\bullet_\ell \triangleq (\bullet_h, \bullet_c)$. The *local state unit element* is $\mathbf{0}_\ell \triangleq (\mathbf{0}_h, \mathbf{0}_c)$. The *partial commutative monoid of local states* is $(\text{LState}, \bullet_\ell, \{\mathbf{0}_\ell\})$.

Given a local state ℓ, we write ℓ^h and ℓ^c for the first and second projections of ℓ, respectively.

Definition 13 (Local assertions)**.** Assume a countably infinite set of *logical variables* $\text{X} \in \text{LVar}$. The set of *logical expressions*, $e \in \text{LExpr}$ is defined by the following inductive grammar, where $v \in \textsc{Val}$, $\mathtt{x} \in \textsc{Var}$ (Def. 1) and $\text{X} \in \text{LVar}$:

$$e ::= v \mid \mathtt{x} \mid \text{X} \mid e + e \mid e * e \mid \ldots$$

Given the set of values Val (Def. 1), assume a set of *logical environments* $\iota \in \text{LEnv} : \text{LVar} \rightharpoonup \textsc{Val}$. Given a stack $\sigma \in \textsc{Stack}$ (Def. 2) and a logical environment $\iota : \text{LEnv}$, the *logical expression evaluation* function, $[\![\cdot]\!]_{(.,.)} : \text{LExpr} \times \textsc{Stack} \times \text{LEnv} \rightharpoonup \textsc{Val}$, is defined inductively over the structure of logical expressions as follows:

$$\begin{array}{rcl} [\![v]\!]_{\iota,\sigma} & \triangleq & v \\ [\![\mathtt{x}]\!]_{\iota,\sigma} & \triangleq & \sigma(v) \\ [\![\text{X}]\!]_{\iota,\sigma} & \triangleq & \iota(\text{X}) \\ [\![e_1 + e_2]\!]_{\iota,\sigma} & \triangleq & [\![e_1]\!]_{\iota,\sigma} + [\![e_2]\!]_{\iota,\sigma} \\ [\![e_1 * e_2]\!]_{\iota,\sigma} & \triangleq & [\![e_1]\!]_{\iota,\sigma} * [\![e_2]\!]_{\iota,\sigma} \end{array}$$

The set of *local assertions*, $p, q \in \text{LAst}$, is defined inductively by the following grammar, where $f \in \textsc{FingerPrint}$ denotes a fingerprint (Def. 5) and $\text{X}, \text{R} \in \text{LVar}$:

$$\begin{array}{c} p, q ::= \text{false} \mid \text{true} \mid p \wedge q \mid p \vee q \mid \exists \text{X}.\ p \\ \mid \text{emp} \mid e \mapsto_f e \mid [\kappa]^{\text{R}} \mid p * q \end{array}$$

Given a logical environment $\iota \in \text{LEnv}$, the *local interpretation function*, $: \text{LAst} \times \text{LEnv} \to \mathcal{P}(\text{LState})$, is defined over the structure of local assertions as follows:

$$\begin{array}{rl} [\![\text{false}]\!]_{\iota,\sigma} & \triangleq \emptyset \\ [\![\text{true}]\!]_{\iota,\sigma} & \triangleq \text{LState} \\ [\![p \wedge q]\!]_{\iota,\sigma} & \triangleq [\![p]\!]_{\iota,\sigma} \cap [\![q]\!]_{\iota,\sigma} \\ [\![p \vee q]\!]_{\iota,\sigma} & \triangleq [\![p]\!]_{\iota,\sigma} \cup [\![q]\!]_{\iota,\sigma} \\ [\![\exists \text{X}.\ p]\!]_{\iota,\sigma} & \triangleq \bigcup_{v \in \textsc{Val}} [\![p]\!]_{\iota[\text{X} \mapsto v],\sigma} \\ [\![\text{emp}]\!]_{\iota,\sigma} & \triangleq \{\mathbf{0}_\ell\} \\ [\![e_1 \mapsto_f e_2]\!]_{\iota,\sigma} & \triangleq \left\{(h, \mathbf{0}_c) \mid \exists a, v.\ [\![e_1]\!]_{\iota,\sigma} = a \wedge [\![e_2]\!]_{\iota,\sigma} = v \wedge \text{dom}(h) = \{a\} \wedge h(a) = (v, f)\right\} \\ [\![[\kappa]^{\text{R}}]\!]_{\iota,\sigma} & \triangleq \left\{(\mathbf{0}_h, c) \mid \exists r.\ \iota(\text{R}) = r \wedge \text{dom}(c) = \{r\} \wedge c(r) = \kappa\right\} \\ [\![p * q]\!]_{\iota,\sigma} & \triangleq \left\{\ell_1 \bullet_\ell \ell_2 \middle| \ell_1 \in [\![p]\!]_{\iota,\sigma} \wedge \ell_2 \in [\![q]\!]_{\iota,\sigma}\right\} \end{array}$$

Observe that program expressions ($\mathbb{E} \in \textsc{Expr}$ in Def. 3) are contained in logical expressions ($e \in \text{LExpr}$ in Def. 13 above). That is, $\textsc{Expr} \subset \text{LExpr}$.

3.2 Rules for Local

The proof rules are standard except TRDEREF and TRMUTATE. The TRDEREF rule add read finger-print in finger-tracking set, only if there is no write finger-print. This is because once a location has been re-written, the rest read are considered as local operations, while the finger-print only records those operations might have effect on global state.

$$\dfrac{\mathtt{x} \notin \mathsf{fv}(\mathbb{E}) \qquad \mathtt{x} \notin \mathsf{fv}(e)}{\vdash \{\mathbb{E} \mapsto_f e\}\ \mathtt{x} := [\mathbb{E}]\ \left\{\mathtt{x} \doteq e * \mathbb{E} \mapsto_{f \overset{\mathtt{r}}{\hookleftarrow}} e\right\}}\ \textsc{TRDeref}$$

$$\dfrac{}{\vdash \{\mathbb{E}_1 \mapsto_f -\}\ [\mathbb{E}_1] := \mathbb{E}_2\ \left\{\mathbb{E}_1 \mapsto_{f \overset{\mathtt{w}}{\hookleftarrow}} \mathbb{E}_2\right\}}\ \textsc{TRMutate}$$

3.3 Global/Program

Definition 14 (Actions)**.** Given the set of local states LSTATE (Def. 12), the set of *actions*, a $\in$ ACTION, is defined as follows:

$$\textsc{Action} \triangleq \left\{ ((h,c),(h',c')) \;\middle|\; \begin{array}{l} ((h,c),(h',c')) \in \textsc{LState} \times \textsc{LState} \\ \wedge\ \mathrm{dom}(h) = \mathrm{dom}(h') \\ \wedge\ \forall a.\ h(a) = (v,f) \Rightarrow \\ \quad \big(h'(a) = (-,f') \wedge f' \subseteq f \overset{\mathtt{w}}{\hookleftarrow}\big) \vee \big(h'(a) = (v,f') \wedge f' = f \overset{\mathtt{r}}{\hookleftarrow}\big) \end{array} \right\}$$

SX: Change the write part to $\subseteq$, because the post condition could be write and read. A more constrained way is to say $f = \emptyset$ to enforce the pre-conditions has no fingerprints.

Given the set of primitive capabilities KAP (Def. 11), the set of *interference environments* is $\mathcal{I} \in \textsc{Inter} \triangleq \textsc{Kap} \rightharpoonup \mathcal{P}(\textsc{Action})$.

SX: Ignore the name clash for now

Definition 15 (Logical states)**.** Assume a partial commutative monoid of (plain) heaps $(\textsc{Heap}, \bullet_h, \mathbf{0}_h)$, where $\textsc{Heap} \triangleq \textsc{Addr} \overset{fin}{\rightharpoonup} \textsc{Val}$, $\bullet_h = \uplus$ and $\mathbf{0}_h = \emptyset$. Then given the partial commutative monoid of capabilities ($\textsc{Cap}$, $\bullet_c$, $\mathbf{0}_c$) in Def. 11, the set of *logical states* is: $l \in \textsc{LGState} \triangleq \textsc{Heap} \times \textsc{Cap}$. The *logical state composition function*, $\bullet_{\mathsf{L}} : \textsc{LGState} \times \textsc{LGState} \rightharpoonup \textsc{LGState}$, is defined component-wise as: $\bullet_{\mathsf{L}} \triangleq (\bullet_h, \bullet_c)$. The *logical state unit element* is $\mathbf{0}_{\mathsf{L}} \triangleq (\mathbf{0}_h, \mathbf{0}_c)$. The *partial commutative monoid of logical states* is $(\textsc{LGState}, \bullet_{\mathsf{L}}, \mathbf{0}_{\mathsf{L}})$.

Definition 16 (Worlds)**.** Given the set of region identifiers REGIONID (Def. 11) and the partial commutative monoid of logical states $(\textsc{LGState}, \bullet_{\mathsf{L}}, \mathbf{0}_{\mathsf{L}})$ in Def. 15, the set of *shared states* is $\textsc{SState} \triangleq \textsc{RegionId} \overset{fin}{\rightharpoonup} \textsc{LGState}$. The *shared state composition function*, $\bullet_{\mathsf{S}} : \textsc{SState} \times \textsc{SState} \rightharpoonup \textsc{SState}$, is defined as: $\bullet_{\mathsf{S}} \triangleq \bullet_{=}$, where for all domains M and all $m, m' \in$ M:

$$m \bullet_{=} m' \triangleq \begin{cases} m & \text{if } m = m' \\ \text{undefined} & \text{otherwise} \end{cases}$$

The *flattening* function for shared states, $\lfloor . \rfloor : \textsc{SState} \rightharpoonup \textsc{LGState}$, is defined as follows, for all $s \in \textsc{SState}$:

$$\lfloor s \rfloor \triangleq \prod_{r \in \mathrm{dom}(s)}^{\bullet_{\mathsf{L}}} s(r)$$

Given the set of timestamps TIMESTAMP (Def. 4), a triple $(l, s, t) \in \textsc{LGState} \times \textsc{SState} \times \textsc{TimeStamp}$ is *well-formed*, written $\mathtt{wf}(l, s, t)$, if and only if:

$$\mathtt{wf}(l, s) \quad \overset{\text{def}}{\iff} \quad \exists h, c.\ l \bullet_{\ell} \lfloor s \rfloor = (h, c) \wedge\ \mathrm{dom}(c) \subseteq \mathrm{dom}(s)$$

AR: Shale, check the last condition of $\mathtt{wf}(.)$ with regards to the timestamps.

SX: Here is my understanding. The first line says a world that is able to collapse down to a TSheap and a time, which is the most important part for proving soundness. The second line means all capabilities must have existed corresponding regions, which is similar to all other logics. The third line I think says a location must at least be initialised? Since we don't have allocation and we make assumption all resources initialised, but I think it is fine we still make this constrains?

The set of *worlds*, $w \in \textsc{World}$, is defined as follows:

$$w \in \textsc{World} \triangleq \left\{(l, s) \;\middle|\; (l, s) \in \textsc{LGState} \times \textsc{SState} \wedge \mathtt{wf}(l, s)\right\}$$

The *world composition function*, $\bullet_w : \textsc{World} \times \textsc{World} \rightharpoonup \textsc{World}$, is defined component-wise as: $\bullet_w \triangleq (\bullet_h, \bullet_{\mathsf{S}})$. The *world unit set* is $\mathbf{0}_w \triangleq \left\{(\mathbf{0}_h, s) \;\middle|\; (\mathbf{0}_h, s) \in \textsc{World}\right\}$. The *partial commutative monoid of worlds* is $(\textsc{World}, \bullet_w, \mathbf{0}_w)$.

Definition 3.1 (Interference)**.** The set of *interference assertions*, $I \in \textsc{IAst}$, are defined by the following grammar:

$$I \triangleq \emptyset \mid \{[\kappa] : \exists \vec{\mathrm{X}}.\, p \rightsquigarrow q\} \cup I$$

Given a logical environment $\iota \in \textsc{LEnv}$ and a stack $\sigma \in \textsc{Stack}$, the *interference interpretation* function, $[\![.]\!]_{(.,.)} : \textsc{IAst} \times \textsc{LEnv} \times \textsc{Stack} \to \textsc{Inter}$, is defined as follows, for all $\kappa \in \textsc{Kap}$:

$$\begin{aligned}
[\![\emptyset]\!]_{\iota,\sigma}(\kappa) &\triangleq \emptyset \\
[\![\{[\kappa] : \exists \vec{\mathrm{X}}.\, p \rightsquigarrow q\} \cup I]\!]_{\iota,\sigma}(\kappa) &\triangleq \left\{(\ell_p, \ell_q) \;\middle|\; \begin{array}{l}(\ell_p, \ell_q) \in \textsc{Action} \wedge \exists r, \vec{v}, \iota'.\, \iota(\mathrm{R}) = r \wedge \iota' = \iota[\vec{\mathrm{X}} \mapsto \vec{v}] \\ \wedge\, \ell_p \in [\![p]\!]_{\iota,\sigma} \wedge \ell_q \in [\![q]\!]_{\iota,\sigma}\end{array}\right\} \cup [\![I]\!]_{\iota,\sigma}(\kappa)
\end{aligned}$$

Definition 17 (Assertions)**.** The set of *assertions*, $P, Q \in \textsc{Ast}$, are defined by the following inductive grammar:

$$\begin{aligned}
P, Q \triangleq\; &\mathrm{false} \mid \mathrm{true} \mid P \wedge Q \mid P \vee Q \mid \exists \mathrm{X}.\, P \\
&\mid \mathrm{emp} \mid e_1 \mapsto e_2 \mid [\kappa]^{\mathrm{R}} \mid \boxed{P}^{\mathrm{R}}_{I} \mid P * Q
\end{aligned}$$

where $\mathrm{X}, \mathrm{R} \in \textsc{LVar}$, $e_1, e_2 \in \textsc{LExpr}$ (Def. 13), $\kappa \in \textsc{Kap}$ (Def. 11) and $I \in \textsc{IAst}$ (Def. 3.1). Given a logical environment $\iota \in \textsc{LEnv}$ and a stack $\sigma \in \textsc{Stack}$, the *assertion interpretation* function, $[\![.]\!]_{(.,.)} : \textsc{Ast} \times \textsc{LEnv} \times \textsc{Stack} \to \textsc{World}$, is defined as follows:

$$\begin{array}{rcl}
[\![\mathrm{false}]\!]_{\iota,\sigma} & \triangleq & \emptyset \\
[\![\mathrm{true}]\!]_{\iota,\sigma} & \triangleq & \textsc{World} \\
[\![\mathrm{emp}]\!]_{\iota,\sigma} & \triangleq & \mathbf{0}_w \\
[\![P \wedge Q]\!]_{\iota,\sigma} & \triangleq & [\![P]\!]_{\iota,\sigma} \cap [\![Q]\!]_{\iota,\sigma} \\
[\![P \vee Q]\!]_{\iota,\sigma} & \triangleq & [\![P]\!]_{\iota,\sigma} \cup [\![Q]\!]_{\iota,\sigma} \\
[\![\exists \mathrm{X}.\, P]\!]_{\iota,\sigma} & \triangleq & \bigcup_{v \in \textsc{Val}} [\![P]\!]_{\iota[\mathrm{X} \mapsto v],\sigma} \\
[\![e_1 \mapsto e_2]\!]_{\iota,\sigma} & \triangleq & \left\{((h, \mathbf{0}_c), s) \;\middle|\; h = \left\{[\![e_1]\!]_{\iota,\sigma} \mapsto [\![e_2]\!]_{\iota,\sigma}\right\} \wedge s \in \textsc{SState}\right\} \\
[\![[\kappa]^{\mathrm{R}}]\!]_{\iota,\sigma} & \triangleq & \left\{((\mathbf{0}_h, c), s) \;\middle|\; \exists r.\, \iota(\mathrm{R}) = r \wedge \mathrm{dom}(c) = \{r\} \wedge c(r) = \kappa\right\} \\
\left[\!\!\left[\boxed{P}^{\mathrm{R}}_{I}\right]\!\!\right]_{\iota,\sigma} & \triangleq & \left\{(\mathbf{0}_{\mathsf{L}}, s) \;\middle|\; \exists r, l.\, \iota(\mathrm{R}) = r \wedge s(r) = (l, [\![I]\!]_{\iota,\sigma}) \wedge (l, s) \in [\![P]\!]^{\mathsf{A}}_{\iota,\sigma}\right\} \\
[\![P * Q]\!]_{\iota,\sigma} & \triangleq & \left\{(w_1 \bullet_w w_2) \;\middle|\; w_1 \in [\![P]\!]_{\iota,\sigma} \wedge w_2 \in [\![Q]\!]_{\iota,\sigma}\right\}
\end{array}$$

with the *auxiliary interpretation function*, $[\![(.)]\!]^{\mathsf{A}}_{(.,.)} : \textsc{Ast} \times \textsc{LEnv} \times \textsc{Stack} \times \to \mathcal{P}(\textsc{LGState} \times \textsc{SState})$, defined as follows:

$$\begin{array}{rcl}
[\![\mathrm{false}]\!]^{\mathsf{A}}_{\iota,\sigma} & \triangleq & \emptyset \\
[\![\mathrm{true}]\!]^{\mathsf{A}}_{\iota,\sigma} & \triangleq & \textsc{LGState} \times \textsc{SState} \\
[\![\mathrm{emp}]\!]^{\mathsf{A}}_{\iota,\sigma} & \triangleq & \left\{(\mathbf{0}_{\mathsf{L}}, s) \;\middle|\; s \in \textsc{SState}\right\} \\
[\![P \wedge Q]\!]^{\mathsf{A}}_{\iota,\sigma} & \triangleq & [\![P]\!]^{\mathsf{A}}_{\iota,\sigma} \cap [\![Q]\!]^{\mathsf{A}}_{\iota,\sigma} \\
[\![P \vee Q]\!]^{\mathsf{A}}_{\iota,\sigma} & \triangleq & [\![P]\!]^{\mathsf{A}}_{\iota,\sigma} \cup [\![Q]\!]^{\mathsf{A}}_{\iota,\sigma} \\
[\![\exists \mathrm{X}.\, P]\!]^{\mathsf{A}}_{\iota,\sigma} & \triangleq & \bigcup_{v \in \textsc{Val}} [\![P]\!]^{\mathsf{A}}_{\iota[\mathrm{X} \mapsto v],\sigma} \\
[\![e_1 \mapsto e_2]\!]^{\mathsf{A}}_{\iota,\sigma} & \triangleq & \left\{((h, \mathbf{0}_c), s) \;\middle|\; h = \left\{[\![e_1]\!]_{\iota,\sigma} \mapsto [\![e_1]\!]_{\iota,\sigma}\right\} \wedge s \in \textsc{SState}\right\} \\
[\![[\kappa]^{\mathrm{R}}]\!]^{\mathsf{A}}_{\iota,\sigma} & \triangleq & \left\{((\mathbf{0}_h, c), s) \;\middle|\; \exists r.\, \iota(\mathrm{R}) = r \wedge \mathrm{dom}(c) = \{r\} \wedge c(r) = \kappa\right\} \\
\left[\!\!\left[\boxed{P}^{\mathrm{R}}_{I}\right]\!\!\right]^{\mathsf{A}}_{\iota,\sigma} & \triangleq & \left\{(l, s) \;\middle|\; (l, s) \in \left[\!\!\left[\boxed{P}^{\mathrm{R}}_{I}\right]\!\!\right]_{\iota,\sigma}\right\} \\
[\![P * Q]\!]^{\mathsf{A}}_{\iota,\sigma} & \triangleq & \left\{((l_1 \bullet_{\mathsf{L}} l_2), (s_1 \bullet_{\mathsf{S}} s_2)) \;\middle|\; (l_1, s_1) \in [\![P]\!]^{\mathsf{A}}_{\iota,\sigma} \wedge (l_2, s_2) \in [\![Q]\!]^{\mathsf{A}}_{\iota,\sigma}\right\}
\end{array}$$

3.4 Merge - shale

Definition 3.2 (Fingerprint heap agreement)**.** Given two fingerprint heaps, the *fingerprint heap agreement* is defined as follows:

$$\begin{array}{rcl}
\mathtt{agree_heap}(h_l, h_r) & \triangleq & \exists h'_l = \mathsf{get_no_write}(h_l), h''_l, h'_r = \mathsf{get_no_write}(h_r), h''_r, h_m. \\
& & \mathtt{ws}(h_l) \cap \mathtt{ws}(h_r) = \emptyset \wedge h'_l = h''_l \bullet_h h_m \wedge h'_r = h''_r \bullet_h h_m \wedge (h_l \bullet_h h_r \bullet_h h_m) \downarrow
\end{array}$$

where,

$$
\begin{array}{rcl}
\mathsf{ws}(h) & \triangleq & \left\{a \middle| \exists f.\, h(a) = (-, f \overset{\mathtt{w}}{\leftrightarrow}, -)\right\} \\
\mathsf{get\_no\_write}(h) & \triangleq & h \setminus \{a \mapsto - | a \in \mathtt{ws}(h)\}
\end{array}
$$

The agreement between two fingerprint heaps means that they write different locations, and for the rest parts, if there is overlap, the overlapped part must be at the same state.

Definition 3.3 (Action agreement)**.** The *action agreement* is defined as follows:

$$
\begin{array}{rcl}
\mathtt{agree\_action}((\ell_l^p, \ell_l^q), (\ell_r^p, \ell_r^q)) & \triangleq & \mathtt{agree\_local}(\ell_l^p, \ell_r^p) \wedge \mathtt{agree\_local}(\ell_l^q, \ell_r^q) \wedge \\
& & \mathtt{agree\_cap\_diff}(\ell_l^p, \ell_r^q, \ell_l^q, \ell_r^p) \\
\mathtt{agree\_local}((h_l, c_l), (h_r, c_r)) & \triangleq & \mathtt{agree\_heap}(h_l, h_r) \wedge (c_l \bullet_c c_r) \downarrow \\
\mathtt{agree\_cap\_diff}((-, c_l^p), (-, c_l^q), (-, c_r^p), (-, c_r^q)) & \triangleq & ((c_l^p \setminus c_l^q) \bullet_c (c_r^p \setminus c_r^q)) \downarrow \wedge ((c_l^p \setminus c_l^q) \bullet_c (c_r^q \setminus c_r^p)) \downarrow \wedge \\
& & ((c_l^q \setminus c_l^p) \bullet_c (c_r^p \setminus c_r^q)) \downarrow \wedge ((c_l^q \setminus c_l^p) \bullet_c (c_r^q \setminus c_r^p)) \downarrow
\end{array}
$$

SX: The first line of agreement on actions is to stop merging of the following two actions:

$$
\begin{array}{l}
[A] : \mathtt{x} \mapsto_\emptyset 0 * \mathtt{y} \mapsto_\emptyset 0 * [C] \rightsquigarrow \mathtt{x} \mapsto_{\{\mathtt{r}\}} 0 * \mathtt{y} \mapsto_{\{\mathtt{w}\}} 1 \\
[B] : \mathtt{x} \mapsto_\emptyset 0 * \mathtt{y} \mapsto_\emptyset 0 * [C] \rightsquigarrow \mathtt{x} \mapsto_{\{\mathtt{w}\}} 1 * \mathtt{y} \mapsto_{\{\mathtt{r}\}} 0
\end{array}
$$

The second line of agreement on actions is to stop merging of the following two actions:

$$
\begin{array}{l}
[A] : \mathtt{x} \mapsto_\emptyset 0 * \mathtt{y} \mapsto_\emptyset 0 \rightsquigarrow \mathtt{x} \mapsto_{\{\mathtt{r}\}} 0 * \mathtt{y} \mapsto_{\{\mathtt{w}\}} 1 * [C] \\
[B] : \mathtt{x} \mapsto_\emptyset 0 * \mathtt{y} \mapsto_\emptyset 0 * [C] \rightsquigarrow \mathtt{x} \mapsto_{\{\mathtt{w}\}} 1 * \mathtt{y} \mapsto_{\{\mathtt{r}\}} 0
\end{array}
$$

Instead of cross-check pre against another's post, we should check the differential.

For two actions to agree, their pre- and post-conditions must agree on the heaps and the capabilities. Additional, the differential of capabilities of these two actions should also agree, so that when merging these two actions, the merged result does not have invalid capabilities transfer.

Definition 3.4 (Merge actions)**.** The *merge actions*, specifically merge the right-hand side to the left-hand side, is defined as follows:

$$
\begin{array}{rcl}
\mathsf{merge\_action} & : & \textsc{Action} \times \textsc{Action} \rightharpoonup \textsc{Action} \\
\mathsf{merge\_action}((\ell_l^p, \ell_l^q), (\ell_r^p, \ell_r^q)) & \triangleq & \begin{cases} (\mathsf{merge\_state}(\ell_l^p, \ell_r^p), \mathsf{merge\_state}(\ell_l^q, \ell_r^q)) & \mathtt{agree\_action}((\ell_l^p, \ell_l^q), (\ell_r^p, \ell_r^q)) \\ \text{undefined} & \text{otherwise} \end{cases}
\end{array}
$$

where,

$$
\begin{array}{rcl}
\mathsf{merge\_state}((h_l, c_l), (h_r, c_r)) & \triangleq & (\mathsf{eager\_merge\_heap}(h_l, h_r), c_l \bullet_c c_r) \\
\mathsf{eager\_merge\_heap}(h_l, h_r) & \triangleq & \lambda a. \begin{cases} h_r(a) & a \in \mathrm{dom}(h_l) \cap \mathtt{ws}(h_r) \\ h_l(a) & a \in \mathrm{dom}(h_l) \setminus \mathtt{ws}(h_r) \\ \text{undefined} & a \notin \mathrm{dom}(h_l) \end{cases}
\end{array}
$$

3.5 Old text from Shale

SX: The nested box is interpreted as flatten box assertion. Yes, r and I should change position.

The box assertion $\boxed{P}_I^r$ asserts part of heap that can be shared, where r is region identifier and I is the interference, i.e. all the possible transitions. The interference is a set of transitions that are labelled by capabilities.

$$
\begin{array}{rcl}
P, Q \in \textsc{Ast} & \triangleq & \text{false} \mid \text{true} \mid P \wedge Q \mid P \vee Q \mid \exists \mathrm{x}.\, P \mid \\
& & \text{emp} \mid e \mapsto e \mid [\kappa]^{\mathrm{R}} \mid \boxed{P}_I^{\mathrm{R}} \mid P * Q
\end{array}
$$

The global assertions are interpreted as a tuple of a world and a time-stamp. A world is a pair of shared state and logical state. The logical state is a pair of time-stamp heap and capability, where capability is a partial finite function from region identifier to a token. The shared state is a partial finite function from region identifier to logical state and its interference.

AR: Again, for the sake of consistency with CAP and CoLoSL, I'd swap the order of logical and shared states in Worlds and have $\textsc{World} \triangleq \textsc{LogicalState} \times \textsc{ShareState}$ instead.

Also I would define worlds as TRIPLES with logical state (capturing privately-owned resources), shared state AND time stamp. Worlds are what global assertions are interpreted over so add the time stamp here and not later. If you do this here, later when you give the semantics of global assertions it is just on worlds and not worlds $\times$ time stamps.

If you choose to define capabilities as I described above, make sure you propagate the notation changes here too.

$$
\begin{aligned}
\phi \in \textsc{LogicalState} &\triangleq \textsc{TSHeap} \times \mathcal{P}(\textsc{Capability}) \\
\delta \in \textsc{ShareState} &\triangleq \textsc{RegionId} \overset{fin}{\rightharpoonup} \textsc{LogicalState} \times \textsc{Inter} \\
w \in \textsc{World} &\triangleq \textsc{LogicalState} \times \textsc{ShareState}
\end{aligned}
$$

The composition of logical states.

$$
\begin{aligned}
\phi_1 \bullet_\phi \phi_2 &\triangleq (\phi_1\downharpoonright_1 \uplus \phi_2\downharpoonright_1, \phi_1\downharpoonright_2 \bullet_{\mathcal{C}} \phi_2\downharpoonright_2) \\
\mathbf{0}_\phi &\triangleq (\emptyset, \emptyset)
\end{aligned}
$$

The composition of capabilities and worlds.

$$
\begin{aligned}
w_1 \bullet_w w_2 &\triangleq \begin{cases} (\delta, \phi_1 \bullet_\phi \phi_2) & w_1 = (\delta, \phi_1) \land w_2 = (\delta, \phi_2) \\ \texttt{undefined} & o.w. \end{cases} \\
\mathbf{0}_w &\triangleq (\emptyset, \mathbf{0}_\phi)
\end{aligned}
$$

AR: The world unit must be a SET with arbitrary state as its shared state component and NOT the empty function: $\mathbf{0}_w \triangleq \{(\delta, \mathbf{0}_\phi) \mid \delta \in \textsc{ShareState}\}$.

The global assertions are interpreted as a set of worlds and their corresponding times.

$$
\begin{aligned}
[\![\text{false}]\!]_{\iota,\sigma} &\triangleq \emptyset \\
[\![\text{true}]\!]_{\iota,\sigma} &\triangleq \mathcal{P}(\textsc{World} \times \textsc{TimeStamp}) \\
[\![\text{emp}]\!]_{\iota,\sigma} &\triangleq \{(\emptyset, t)\} \\
[\![\mathbb{E}_1 \mapsto \mathbb{E}_2]\!]_{\iota,\sigma} &\triangleq \left\{ ((\emptyset, (H, \emptyset)), t) \middle| \begin{array}{l} \exists a = [\![\mathbb{E}_1]\!]_{\iota,\sigma}, v = [\![\mathbb{E}_2]\!]_{\iota,\sigma}, t'.\, t' \le t \land \mathrm{dom}(H) = \{a\} \land H(a)(t') = v \\ H(a)(t') = (v, -, -) \land \forall t'' \in (t', t).\, H(a)(t'') \uparrow \end{array} \right\} \\
\left[\!\left[\boxed{P}_{r}^{I} \right]\!\right]_{\iota,\sigma} &\triangleq \left\{ ((\delta, \phi), t) \middle| \exists \delta', \phi'.\, ((\delta', \phi \bullet_\phi \phi'), t) \in [\![P]\!]_{\iota,\sigma,t} \land \delta = \delta' \uplus \left\{ r \mapsto (\phi', [\![I]\!]_{\iota,\sigma}) \right\} \right\} \\
[\![t]\!]_{\iota,\sigma} &\triangleq \left\{ ((\delta, (H, \mathcal{C})), t) \middle| \delta = \emptyset \land H = \emptyset \land \mathcal{C} = [\![t]\!]_{\iota,\sigma} \right\} \\
[\![P * Q]\!]_{\iota,\sigma} &\triangleq \left\{ (w_1 \bullet_w w_2, t) \middle| (w_1, t) \in [\![P]\!]_{\iota,\sigma,t} \land (w_2, t) \in [\![Q]\!]_{\iota,\sigma,t} \right\} \\
[\![P \land Q]\!]_{\iota,\sigma} &\triangleq [\![P]\!]_{\iota,\sigma} \cap [\![Q]\!]_{\iota,\sigma} \\
[\![P \lor Q]\!]_{\iota,\sigma} &\triangleq [\![P]\!]_{\iota,\sigma} \cup [\![Q]\!]_{\iota,\sigma} \\
[\![\exists \mathrm{x}.\, P]\!]_{\iota,\sigma} &\triangleq [\![P]\!]_{\iota[\mathrm{x} \mapsto v],\sigma}
\end{aligned}
$$

AR: If you redefine worlds as described above, make sure you propagate the changes here. For instance, you will instead have: $[\![\text{true}]\!]_{\iota,\sigma} \triangleq \textsc{World}$.

There are a few issues:

$$
\begin{aligned}
[\![\text{emp}]\!]_{\iota,\sigma} &\triangleq \{(w, t) \mid w \in \mathbf{0}_w\} \\
[\![\exists \mathrm{x}.\, P]\!]_{\iota,\sigma} &\triangleq \bigcup_{v \in \textsc{Val}} [\![P]\!]_{\iota[\mathrm{x} \mapsto v],\sigma}
\end{aligned}
$$

The boxed assertion must have an EMPTY local component. Also you must allow the region identifier to be a logical variable (this is useful when you existentially quantify them upon creation) and look it up in the logical environment.

We define *merge* with respect to two heaps associated with their read sets and write sets. The operation $\lhd$ eagerly merge the left-hand side to the right-hand side. This means that, first, the domain of the result is the same as the domain of left-hand side. Second, the result takes left-hand side but propagate all the write effects from right-hand side.

AR: I don't quite understand this definition. First, what do you mean by the domain of h? h is a triple and not a function.
Second, What is the intuition behind $\lhd$? When is it used? I think it is used to calculate the postcondition of actions?

$$
\begin{array}{rcl}
\mathrm{dom}(h) & \triangleq & \mathrm{dom}(h\lfloor_1) \\
h \lhd h' & \triangleq & \left(\lambda a. \begin{cases} h'(a) & a \in ws' \cap \mathrm{dom}(h) \\ h(a) & a \notin ws' \land a \in \mathrm{dom}(h) \\ \texttt{undefined} & o.w. \end{cases}, \; rs \cup (\mathrm{dom}(h) \cap rs'), \; ws \cup (\mathrm{dom}(h) \cap ws') \right) \\
& & \texttt{where } h = (h, rs, ws) \land h' = (h', rs', ws')
\end{array}
$$

The first element, also pre-condition $p^\uparrow$, only contains empty label, $- \mapsto_\emptyset -$, because the semantics requires the pre-condition interpreted as a heap with empty write set and empty read set. The second element, also called post-condition $q^\uparrow$, should contain the same resources as pre-condition. If a location has only been read, the value should remain the same pre-condition.

SX: For resource moving in can be tagged with write but don't know how to tag resource move out. This is not very right.

The PRCommit rule looks the same but the repartition is much more complicated by merging all possible transitions from the environment that are allowed to run concurrently and commit in the same time. Note that after the merging, one still need to check stabilisation.

$$
\frac{\vdash \{p\} \; \mathbb{T} \; \{q\} \qquad \vdash P \Rrightarrow^{\{p\}\{q\}} Q}{\vdash \{P\} \; [\mathbb{T}] \; \{Q\}} \; \textsc{PRCommit}
$$

Given the *merge*for h, we can define *merge* on transitions. The predicate `noFingerPrint` asserts the read set and write set are empty sets, and `noWriteConflict` asserts the write sets are disjointed. The predicate `agree` has similar meaning as overlapped separation found in CoLoSL, which means that the state of the overlapped parts must agree. To merge two transactions, if the pre-conditions agree and both of them have no fingerprint, and if the post-conditions agree and they write to different locations, the merged result is a set of transitions including the left-hand side and another transition by merging the post-condition of right-hand side to left-hand side. Otherwise, the merge only returns a singleton set of left-hand side.

$$
\begin{array}{rcl}
\texttt{noFingerPrint}(h) & \triangleq & \exists rs, ws, . \, h = (-, rs, ws) \land rs = ws = \emptyset \\
\texttt{noWriteConflict}(h_l, h_r) & \triangleq & \exists ws_l = h_l\lfloor_3, ws_r = h_r\lfloor_3 . \, ws_l \cap ws_r = \emptyset \\
\texttt{agreeState}(h_l, h_r) & \triangleq & \exists h'_l, h_m, h'_r, h. \, h_l = h'_l \bullet_h h_m \land h_r = h'_r \bullet_h h_m \land h = h'_l \bullet_h h_m \bullet_h h'_r \\
\texttt{agreeCapability}(\mathcal{C}_l, \mathcal{C}_r) & \triangleq & \exists \mathcal{C}'_l, \mathcal{C}_m, \mathcal{C}'_r, h. \, \mathcal{C}_l = \mathcal{C}'_l \bullet_\mathcal{C} \mathcal{C}_m \land \mathcal{C}_r = \mathcal{C}'_r \bullet_\mathcal{C} \mathcal{C}_m \land \mathcal{C} = \mathcal{C}'_l \bullet_\mathcal{C} \mathcal{C}_m \bullet_\mathcal{C} \mathcal{C}'_r \\
\texttt{agree}(\hat{\hat{h}}_l, \hat{\hat{h}}_r) & \triangleq & \texttt{agreeState}(\hat{\hat{h}}_l\lfloor_1, \hat{\hat{h}}_r\lfloor_1) \land \texttt{agreeCapability}(\hat{\hat{h}}_l\lfloor_2, \hat{\hat{h}}_r\lfloor_2) \\
(\hat{\hat{h}}_p, \hat{\hat{h}}_q) \blacktriangleleft (\hat{\hat{h}}'_p, \hat{\hat{h}}'_q) & \triangleq & \left\{ (\hat{\hat{h}}_p, \hat{\hat{h}}_q) \right\} \cup \left\{ (\hat{\hat{h}}_p, (h_q \lhd h'_q, \mathcal{C}_q \bullet_\mathcal{C} \mathcal{C}'_q)) \middle| \begin{array}{l} \texttt{noWriteConflict}(h'_p, h'_q) \land \\ \texttt{agree}(\hat{\hat{h}}_p, \hat{\hat{h}}_q) \land \texttt{agree}(\hat{\hat{h}}'_p, \hat{\hat{h}}'_q) \end{array} \right\}
\end{array}
$$

The repartition is redefined by adding a merging process. The notation $\lfloor w \rfloor$ collapses a world to a time-stamp heap and capabilities by compositing the private logical state ϕ' and all the shared logical states $\delta(r_i)$. To simplify we reuse the same notation for $\lfloor (w, t) \rfloor$, it further collapses the time-stamp heap to a plain heap by taking a snapshot at the time t. The lift of a plain heap is a set of all possible worlds with their times that collapse to the plain heap with any possible capabilities, and similarly the lift of a plain heap with its read and write sets is the lift of the plain heap but lose the information about read and write sets. Given that $\hat{\hat{h}}$ talks about interference, either pre- or post-condition, of a certain region, the lift of $\hat{\hat{h}}$ is all possible worlds with their times where the lift of the plain heap, i.e. the first projection of $\hat{\hat{h}}$ is subset of the time-stamp heap corresponding to the region r and the capabilities of $\hat{\hat{h}}$, the second projection, is the subset of the capabilities of region r.

The repartition means, for all possible world and time, (w, t), that satisfies global assertion P, there exist a heap with its read set and write set, h, that satisfies local assertion p, h' for assertion q and a transition $(\hat{\hat{h}}, \hat{\hat{h}}')$ that is allowed by the guarantees. The h should agree with $\hat{\hat{h}}$, which means $h = \hat{\hat{h}}\lfloor_1$ and similarly $h' = \hat{\hat{h}}'\lfloor_1$. Also, the (w, t) corresponding to P should agree with the pre-condition of the transition, i.e. $(w, t) \in \left\lceil \hat{\hat{h}} \right\rceil$. Then for all possible merging between transition $(\hat{\hat{h}}, \hat{\hat{h}}')$ and relies, there must exist a corresponding world and its time satisfies the global assertion Q. It means that the world and its time (w', t') is in the lift of post-condition of the merging result $\hat{\hat{h}}''$. At last the worlds w and w' should be balanced, meaning no creation or destruction of resources.

$$
\begin{array}{rcl}
\lfloor w \rfloor & \triangleq & (H, \mathcal{C}) \texttt{ where } \exists \delta, \phi, \phi', r_0, \ldots, r_n. \\
& & \qquad w = (\delta, \phi') \wedge \{r_0, \ldots, r_n\} = \mathrm{dom}(\delta) \wedge \phi = \phi' \bullet_\phi \delta(r_1) \bullet_\phi \ldots \bullet_\phi \delta(r_n) \\
& & \qquad H = \phi\lfloor_1 \wedge \mathcal{C} = \phi\lfloor_2 \\
\lfloor (w, t) \rfloor & \triangleq & (h, \mathcal{C}) \texttt{ where } h = \mathsf{startstate}(\lfloor w \rfloor\lfloor_1, t) \wedge \mathcal{C} = \lfloor w \rfloor\lfloor_2 \\
\left\lfloor \hat{\hat{h}} \right\rfloor & \triangleq & (h, \mathcal{C}) \texttt{ where } h = h\lfloor_1\lfloor_1 \wedge \mathcal{C} = \hat{\hat{h}}\lfloor_2 \\
\texttt{balance}(w_1, w_2) & \triangleq & \exists H_1, \mathcal{C}_1, H_2, \mathcal{C}_2.\, (H_1, \mathcal{C}_1) = \lfloor w_1 \rfloor \wedge (H_2, \mathcal{C}_2) = \lfloor w_2 \rfloor \wedge \mathrm{dom}(H_1) = \mathrm{dom}(H_2) \wedge \mathcal{C}_1 = \mathcal{C}_2 \\
\vdash P \Rrightarrow^{\{p\}\{q\}} Q & \iff & \forall \iota, \sigma, w, t.\, (w, t) \in \llbracket P \rrbracket_{\iota, \sigma}. \\
& & \implies \exists \iota', \sigma', h \in \llbracket p \rrbracket_{\iota, \sigma}, h' \in \llbracket q \rrbracket_{\iota', \sigma'}, \hat{\hat{h}}, \hat{\hat{h}}', \hat{\hat{h}}_r. \\
& & \qquad (\hat{\hat{h}}, \hat{\hat{h}}') \in G(w) \wedge h = \hat{\hat{h}}\lfloor_1 \wedge h' = \hat{\hat{h}}'\lfloor_1 \wedge \lfloor (w, t) \rfloor = \left\lfloor \hat{\hat{h}} \bullet_{\hat{\hat{h}}} \hat{\hat{h}}_r \right\rfloor \\
& & \implies \forall \hat{\hat{h}}''.\, (-, \hat{\hat{h}}'') \in ((\hat{\hat{h}}, \hat{\hat{h}}') \blacktriangleleft R(w)) \\
& & \exists w', t'.\, t < ts' \wedge (w', t') \in \llbracket Q \rrbracket_{\iota', \sigma'} \wedge \lfloor (w', t') \rfloor = \left\lfloor \hat{\hat{h}}'' \bullet_{\hat{\hat{h}}} \hat{\hat{h}}_r \right\rfloor \wedge \texttt{balance}(w, w')
\end{array}
$$

The relies and guarantees.

$$
\begin{array}{rcl}
G & \triangleq & \lambda w. \left\{ (\hat{\hat{h}}, \hat{\hat{h}}') \middle| \forall c \in w\lfloor_2\lfloor_2 \wedge (\hat{\hat{h}}, \hat{\hat{h}}') \in w\lfloor_1 (c\lfloor_1)\lfloor_2 (c) \right\} \\
R & \triangleq & \lambda w. \left\{ (\hat{\hat{h}}, \hat{\hat{h}}') \middle| \forall c.\, \exists c' \in w\lfloor_2\lfloor_2.\, c\lfloor_1 = c'\lfloor_1 \wedge ((c\lfloor_2, c\lfloor_3) \bullet_\Gamma (c'\lfloor_2, c'\lfloor_3)) \downarrow \wedge (\hat{\hat{h}}, \hat{\hat{h}}') \in w\lfloor_1 (c\lfloor_1)\lfloor_2 (c') \right\}
\end{array}
$$

4 temp

To merge two actions $p \rightsquigarrow q$ and $p' \rightsquigarrow q'$, it requires the pre-conditions agrees, which means that if they describe some common heaps, the state of the common part should be consistent. Then if the post-conditions write to different locations, the *merge* (to the left) operator $\lhd$ returns a new action where the effect of right hands side propagate to the left hand side.

SX: Not sure how to do it properly considering there is $\vec{x}$ binder and extension $\vec{y}$

.

$$
\begin{array}{rcl}
\mathsf{writeSet}(\mathrm{X} \mapsto_{f \cup \{\mathtt{w}\}} -) & \triangleq & \{\mathrm{X}\} \\
\mathsf{writeSet}(\mathrm{X} \mapsto_{f \setminus \{\mathtt{w}\}} -) & \triangleq & \emptyset \\
\mathsf{writeSet}(p * q) & \triangleq & \mathsf{writeSet}(p) \uplus \mathsf{writeSet}(q) \\
p \blacktriangleleft q & \triangleq & \begin{cases} (p' \blacktriangleleft q') * \mathrm{X} \mapsto_{f \cup \{\mathtt{w}\}} - & (p = p' * \mathrm{X} \mapsto_{\_} -) \wedge (q = q' * \mathrm{X} \mapsto_{f \cup \{\mathtt{w}\}} -) \\ p & o.w. \end{cases} \\
\mathsf{agree}(p, q) & \triangleq & \exists p', q', r, m.\, (p = p * r) \wedge (q = q' * r) \wedge (m = p' * q' * r) \\
(p \rightsquigarrow q) \lhd (p' \rightsquigarrow q') & \triangleq & \{p \rightsquigarrow q\} \cup \{p \rightsquigarrow (q \blacktriangleleft q') | \mathsf{agree}(p, p') \wedge \mathsf{writeSet}(q) \cap \mathsf{writeSet}(q') = \emptyset\}
\end{array}
$$

5 blablabla

We use $a_{\_}$ to denote a location in either global or local heap.

$$\{a_x \mapsto 0 * a_y \mapsto 0\}$$

$$\begin{array}{l}
R:\ a_x \mapsto_r _ * a_y \mapsto_r 0 \leadsto a_x \mapsto_w 1 * a_y \mapsto_r 0\\
\{a_x \mapsto 0 * a_y \mapsto 0 \vee a_x \mapsto 1 * a_y \mapsto 0\}\\
\left[\begin{array}{l}
\left\{\begin{array}{l} a_x \mapsto_r 0 * a_y \mapsto_r 0 \vee \\ a_x \mapsto_r 1 * a_y \mapsto_r 0 \end{array}\right\}\\
\mathtt{x} := [a_x];\\
\left\{\begin{array}{l} a_x \mapsto_r 0 * a_y \mapsto_r 0 \wedge \mathtt{x} = 0 \vee \\ a_x \mapsto_r 1 * a_y \mapsto_r 0 \wedge \mathtt{x} = 1 \end{array}\right\}\\
\mathtt{if}\ (\mathtt{x} = 0)\\
\quad [a_y] := 1;\\
\left\{\begin{array}{l} a_x \mapsto_r 0 * a_y \mapsto_w 1 \wedge \mathtt{x} = 0 \vee \\ a_x \mapsto_r 1 * a_y \mapsto_r 0 \wedge \mathtt{x} = 1 \end{array}\right\}\\
MERGE\\
\left\{\begin{array}{l} a_x \mapsto_r 0 * a_y \mapsto_w 1 \vee \\ a_x \mapsto_r 1 * a_y \mapsto_r 0 \vee \\ a_x \mapsto_w 1 * a_y \mapsto_w 1 \vee \\ a_x \mapsto_w 1 * a_y \mapsto_r 0 \end{array}\right\}
\end{array}\right]\\
\left\{\begin{array}{l} a_x \mapsto 0 * a_y \mapsto 1 \vee \\ a_x \mapsto 1 * a_y \mapsto 0 \vee \\ a_x \mapsto 1 * a_y \mapsto 1 \end{array}\right\}
\end{array}
\Bigg\|
\begin{array}{l}
R:\ a_x \mapsto_r 0 * a_y \mapsto_r _ \leadsto a_x \mapsto_r 0 * a_y \mapsto_w 1\\
\{a_x \mapsto 0 * a_y \mapsto 0 \vee a_x \mapsto 0 * a_y \mapsto 1\}\\
\left[\begin{array}{l}
\left\{\begin{array}{l} a_x \mapsto_r 0 * a_y \mapsto_r 0 \vee \\ a_x \mapsto_r 0 * a_y \mapsto_r 1 \end{array}\right\}\\
\mathtt{y} := [a_y];\\
\mathtt{if}\ (\mathtt{y} = 0)\\
\quad [a_x] := 1;\\
\left\{\begin{array}{l} a_x \mapsto_w 1 * a_y \mapsto_r 0 \vee \\ a_x \mapsto_r 0 * a_y \mapsto_r 1 \end{array}\right\}\\
MERGE\\
\left\{\begin{array}{l} a_x \mapsto_w 1 * a_y \mapsto_r 0 \vee \\ a_x \mapsto_r 0 * a_y \mapsto_r 1 \vee \\ a_x \mapsto_w 1 * a_y \mapsto_w 1 \vee \\ a_x \mapsto_r 0 * a_y \mapsto_w 1 \end{array}\right\}
\end{array}\right]\\
\left\{\begin{array}{l} a_x \mapsto 0 * a_y \mapsto 1 \vee \\ a_x \mapsto 1 * a_y \mapsto 0 \vee \\ a_x \mapsto 1 * a_y \mapsto 1 \end{array}\right\}
\end{array}$$

$$\{a_x \mapsto 0 * a_y \mapsto 1 \vee a_x \mapsto 1 * a_y \mapsto 0 \vee a_x \mapsto 1 * a_y \mapsto 1\}$$

***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.

$$\begin{array}{l}
\quad a_y \mapsto_r _ \leadsto a_y \mapsto_w 1\\
R:\ a_x \mapsto_r x * a_y \mapsto_r y * a_{tx2} \mapsto_r _ * a_{ty2} \mapsto_r _\\
\quad \leadsto a_x \mapsto_r x * a_y \mapsto_r y * a_{tx2} \mapsto_w x * a_{ty2} \mapsto_w y\\
\left\{\begin{array}{l} a_x \mapsto 0 * a_y \mapsto 0 * a_{tx1} \mapsto 0 * a_{ty1} \mapsto 0 \vee \\ a_x \mapsto 0 * a_y \mapsto 1 * a_{tx1} \mapsto 0 * a_{ty1} \mapsto 0 \end{array}\right\}\\
\qquad [[a_x] := 1;]\\
\left\{\begin{array}{l} a_x \mapsto 1 * a_y \mapsto 0 * a_{tx1} \mapsto 0 * a_{ty1} \mapsto 0 \vee \\ a_x \mapsto 1 * a_y \mapsto 1 * a_{tx1} \mapsto 0 * a_{ty1} \mapsto 0 \end{array}\right\}\\
\left[\begin{array}{l}
\left\{\begin{array}{l} a_x \mapsto_r 1 * a_y \mapsto_r 0 * a_{tx1} \mapsto_r 0 * a_{ty1} \mapsto_r 0 \vee \\ a_x \mapsto_r 1 * a_y \mapsto_r 1 * a_{tx1} \mapsto_r 0 * a_{ty1} \mapsto_r 0 \end{array}\right\}\\
\mathtt{x} := [a_x];\\
[a_{tx1}] := \mathtt{x};\\
\mathtt{y} := [a_y];\\
[a_{ty1}] := \mathtt{y};\\
\left\{\begin{array}{l} a_x \mapsto_r 1 * a_y \mapsto_r 0 * a_{tx1} \mapsto_w 1 * a_{ty1} \mapsto_w 0 \vee \\ a_x \mapsto_r 1 * a_y \mapsto_r 1 * a_{tx1} \mapsto_w 1 * a_{ty1} \mapsto_w 1 \end{array}\right\}\\
MERGE\\
\left\{\begin{array}{l} a_x \mapsto_r 1 * a_y \mapsto_r 0 * a_{tx1} \mapsto_w 1 * a_{ty1} \mapsto_w 0 \vee \\ a_x \mapsto_r 1 * a_y \mapsto_r 1 * a_{tx1} \mapsto_w 1 * a_{ty1} \mapsto_w 1 \vee \\ a_x \mapsto_r 1 * a_y \mapsto_w 1 * a_{tx1} \mapsto_w 1 * a_{ty1} \mapsto_w 0 \end{array}\right\}
\end{array}\right]\\
\left\{\begin{array}{l} a_x \mapsto 1 * a_y \mapsto 0 * a_{tx1} \mapsto 1 * a_{ty1} \mapsto 0 \vee \\ a_x \mapsto 1 * a_y \mapsto 1 * a_{tx1} \mapsto 1 * a_{ty1} \mapsto 1 \vee \\ a_x \mapsto 1 * a_y \mapsto 1 * a_{tx1} \mapsto 1 * a_{ty1} \mapsto 0 \end{array}\right\}
\end{array}
\Bigg\|
\begin{array}{l}
\quad a_x \mapsto_r _ \leadsto a_x \mapsto_w 1\\
R:\ a_x \mapsto_r x * a_y \mapsto_r y * a_{tx1} \mapsto_r _ * a_{ty1} \mapsto_r _\\
\quad \leadsto a_x \mapsto_r x * a_y \mapsto_r y * a_{tx1} \mapsto_w x * a_{ty1} \mapsto_w y\\
\left\{\begin{array}{l} a_x \mapsto 0 * a_y \mapsto 0 * a_{tx2} \mapsto 0 * a_{ty2} \mapsto 0 \vee \\ a_x \mapsto 1 * a_y \mapsto 0 * a_{tx2} \mapsto 0 * a_{ty2} \mapsto 0 \end{array}\right\}\\
\qquad [[a_y] := 1;]\\
\left\{\begin{array}{l} a_x \mapsto 0 * a_y \mapsto 1 * a_{tx2} \mapsto 0 * a_{ty2} \mapsto 0 \vee \\ a_x \mapsto 1 * a_y \mapsto 1 * a_{tx2} \mapsto 0 * a_{ty2} \mapsto 0 \end{array}\right\}\\
\left[\begin{array}{l}
\mathtt{x} := [a_x];\\
[a_{tx2}] := \mathtt{x};\\
\mathtt{y} := [a_y];\\
[a_{ty2}] := \mathtt{y};\\
MERGE\\
\left\{\begin{array}{l} a_x \mapsto_r 0 * a_y \mapsto_r 1 * a_{tx2} \mapsto_w 0 * a_{ty2} \mapsto_w 1 \vee \\ a_x \mapsto_r 1 * a_y \mapsto_r 1 * a_{tx2} \mapsto_w 1 * a_{ty2} \mapsto_w 1 \vee \\ a_x \mapsto_w 1 * a_y \mapsto_r 1 * a_{tx2} \mapsto_w 0 * a_{ty2} \mapsto_w 1 \end{array}\right\}
\end{array}\right]\\
\left\{\begin{array}{l} a_x \mapsto 0 * a_y \mapsto 1 * a_{tx2} \mapsto 0 * a_{ty2} \mapsto 1 \vee \\ a_x \mapsto 1 * a_y \mapsto 1 * a_{tx2} \mapsto 1 * a_{ty2} \mapsto 1 \vee \\ a_x \mapsto 1 * a_y \mapsto 1 * a_{tx2} \mapsto 0 * a_{ty2} \mapsto 1 \end{array}\right\}
\end{array}$$

$$\left\{\begin{array}{l} a_x \mapsto 1 * a_y \mapsto 1 * \\ \left(\begin{array}{l} a_{tx1} \mapsto 1 * a_{ty1} \mapsto 0 * a_{tx2} \mapsto 0 * a_{ty2} \mapsto 1 \vee \\ a_{tx1} \mapsto 1 * a_{ty1} \mapsto 0 * a_{tx2} \mapsto 1 * a_{ty2} \mapsto 1 \vee \\ a_{tx1} \mapsto 1 * a_{ty1} \mapsto 1 * a_{tx2} \mapsto 0 * a_{ty2} \mapsto 1 \vee \\ a_{tx1} \mapsto 1 * a_{ty1} \mapsto 1 * a_{tx2} \mapsto 1 * a_{ty2} \mapsto 1 \end{array}\right) \end{array}\right\}$$

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.

$$
\begin{array}{l}
a_y \mapsto_r \_ \rightsquigarrow a_y \mapsto_w 1 \\
R: a_x \mapsto_r x * a_y \mapsto_r y * a_{tx2} \mapsto_r \_ * a_{ty2} \mapsto_r \_ \\
\quad \rightsquigarrow a_x \mapsto_r x * a_y \mapsto_r y * a_{tx2} \mapsto_w x * a_{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)\} \\
[a_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} := [a_x]; \\
[a_{tx1}] := \mathtt{x}; \\
\mathtt{y} := [a_y]; \\
[a_{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\}
\end{array}
\;\Bigg\|\;
\begin{array}{l}
a_x \mapsto_r \_ \rightsquigarrow a_x \mapsto_w 1 \\
R: a_x \mapsto_r x * a_y \mapsto_r y * a_{tx1} \mapsto_r \_ * a_{ty1} \mapsto_r \_ \\
\quad \rightsquigarrow a_x \mapsto_r x * a_y \mapsto_r y * a_{tx1} \mapsto_w x * a_{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)\} \\
[a_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} := [a_x]; \\
[a_{tx2}] := \mathtt{x}; \\
\mathtt{y} := [a_y]; \\
[a_{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\}
\end{array}
$$

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

$$\left[\begin{array}{l}[a_x] := 1; \\ [a_z] := 1;\end{array}\right] \Big\| \left[\begin{array}{l}[a_y] := 2; \\ [a_z] := 2;\end{array}\right]$$

A Proof of semantics

Lemma A.1. Threads can only write to values that are originally undefined,

$$\forall \sigma, \sigma', H, H', t, t', \mathbb{P}^{\uparrow}, \mathbb{P}^{\uparrow'}, \iota, a, t.\, (\sigma, H, t), \mathbb{P}^{\uparrow} \overset{\iota}{\rightsquigarrow}_t (\sigma', H', t'), \mathbb{P}^{\uparrow'} \implies H(a)(t) \uparrow \vee H(a)(t) = H'(a)(t)$$

Proof. Prove by structural induction on the semantics. For base cases PChoiseLeft, PChoiseRight, PLoop, PSeqSkip, PPar and PWait, it is trivial because those rules do not change the state of time-stamp heap H. For base case Commit, the new state $H' = \mathsf{commitTrans}(H, h_s, h_e, ws, rs, \alpha, t_s, t_e)$ for some $h_s, h_e, ws, rs, \alpha, t_s, t_e$. It only changes the values corresponding to times t_s or t_e of locations that are included in either the read set rs or the write set ws. Those must be undefined in the original time-stamp heap H, because of the constrain `allowcommit`, especially the `wellformhist`. For the inductive case PSeq, prove directly by the inductive hypothesis. ☐

Lemma A.2. All the reads/starts operations of a transaction happen before all the writes/ends. Formally,

$$\forall \sigma, \sigma', H, H', t, t', \mathbb{P}^{\uparrow}, \mathbb{P}^{\uparrow'}, \iota.\, \mathtt{rw}(H) \wedge (\sigma, H, t), \mathbb{P}^{\uparrow} \overset{\iota}{\rightsquigarrow}_t (\sigma', H', t'), \mathbb{P}^{\uparrow'} \implies \mathtt{rw}(H')$$

Where,

$$\mathtt{rw}(H) \triangleq \forall a, a', t, t', e \in \{\mathtt{S}, \mathtt{R}\}, e' \in \{\mathtt{E}, \mathtt{W}\}\, \alpha.\, H(a)(t) = (-, e, \alpha) \wedge H(a')(t') = (-, e', \alpha) \implies t < t'$$

Proof. Prove by structural induction on the semantics. For base cases PChoiseLeft, PChoiseRight, PLoop, PSeqSkip, PPar and PWait, it is trivial because those rules do not change the state of time-stamp heap H. For base case Commit, the new state $H' = \mathsf{commitTrans}(H, h_s, h_e, ws, rs, \alpha, t_s, t_e)$ for some $h_s, h_e, ws, rs, \alpha, t_s, t_e$. Because the functions associates R, S to time t_s, and W, E to t_e, and the fact that $t_s < t_e$, so $\mathtt{rw}(H')$ holds. For the inductive case PSeq, prove directly by the inductive hypothesis. ☐

Lemma A.3. All the reads/starts operations among all locations of a transaction happen in the same time, so do all the writes/ends operations.

$$\forall \sigma, \sigma', H, H', t, t', \mathbb{P}^{\uparrow}, \mathbb{P}^{\uparrow'}, \iota.\, \mathtt{atom}(H) \wedge (\sigma, H, t), \mathbb{P}^{\uparrow} \overset{\iota}{\rightsquigarrow}_t (\sigma', H', t'), \mathbb{P}^{\uparrow'} \implies \mathtt{atom}(H')$$

Where,

$$\mathtt{atom}(H) \triangleq \forall a, a', t, t', \alpha, e, e'.\, H(a)(t)(-, e, \alpha) \wedge H(a')(t') = (-, e', \alpha) \wedge (e, e' \in \{\mathtt{S}, \mathtt{R}\} \vee e, e' \in \{\mathtt{E}, \mathtt{W}\}) \implies t = t'$$

Proof. Prove by structural induction on the semantics. For base cases PCHOISELEFT, PCHOISERIGHT, PLOOP, PSEQSKIP, PPAR and PWAIT, it is trivial because those rules do not change the state of time-stamp heap H. For base case COMMIT, the new state $H' = \mathsf{commitTrans}(H, h_s, h_e, ws, rs, \alpha, t_s, t_e)$ for some $h_s, h_e, ws, rs, \alpha, t_s, t_e$. Because the functions associates all reads/starts operations, $\mathtt{R}$ and $\mathtt{S}$, of all locations to the same time t_s, and writes/ends operations of all locations to the same time t_e, so $\mathtt{atom}(H')$ holds. For the inductive case PSEQ, prove directly by the inductive hypothesis. ☐

To define trace and then program order, we first extend the labels used in the semantics. Each label has one extra parameter to record the thread that preforms the step, so the label looks like $\mathtt{cmt}(i, \alpha)$, $\mathtt{fork}(i, i', \mathbb{P})$ and $\mathtt{join}(i, i', t)$.

Definition A.4 (Traces)**.** Given a initial state H and η, a trace θ is define as a tuple $(\mathcal{L}, <)$ that satisfies predicate $\mathtt{trace}(H, \eta, \mathcal{L}, <, n)$, for some number n.

$$\begin{array}{rcl}
\mathsf{traces}(H, \eta, 0) & \triangleq & \{(\emptyset, \emptyset, H, \eta)\} \\
\mathsf{traces}(H, \eta, n) & \triangleq & \left\{(\mathcal{L}, <, H', \eta') \middle| (H'', \eta'') \overset{\mathtt{id}}{\rightsquigarrow}_g (H', \eta') \wedge (\mathcal{L}, <, H'', \eta'') \in \mathsf{trace}(H, \eta, \mathcal{L}, <, n-1)\right\} \\
& & \uplus \left\{(\mathcal{L} \uplus \{\iota\}, < \uplus \{(\iota', \iota) | \iota' \in \mathcal{L}\}, H', \eta') \middle| \begin{array}{l} (H'', \eta'') \overset{\iota}{\rightsquigarrow}_g (H', \eta') \wedge \iota \neq \mathtt{id} \wedge \\ (\mathcal{L}, <, H'', \eta'') \in \mathsf{trace}(H, \eta, \mathcal{L}, <, n-1) \end{array}\right\}
\end{array}$$

Definition A.5 (Program Order)**.**

$$\begin{array}{rcl}
\mathsf{programOrder}(\mathcal{L}, <) & \triangleq & (\mathcal{T}, \mathsf{po}) \\
\mathcal{T} & \equiv & \{\alpha | \mathtt{cmt}(i, \alpha) \in \mathcal{L}\} \\
\mathsf{po} & \equiv & \{(\alpha, \alpha') | \mathtt{cmt}(i, \alpha) < \mathtt{cmt}(i, \alpha')\} \uplus \{(\alpha, \alpha') | \mathtt{cmt}(i, \alpha) < \mathtt{fork}(i, i', -) < \mathtt{cmt}(i', \alpha')\} \\
& & \uplus \{(\alpha, \alpha') | \mathtt{cmt}(i, \alpha) < \mathtt{join}(i', i, -) < \mathtt{cmt}(i', \alpha')\}
\end{array}$$

Definition A.6. Visibility and potential arbitration relations.

$$\begin{array}{rcl}
\mathsf{graph}(\mathcal{T}, H) & \triangleq & (\mathtt{vis}, \mathtt{tar}) \\
\mathtt{vis} & \equiv & \left\{(\alpha, \alpha') \in \mathcal{T} \middle| \begin{array}{l} \exists a, a', t, t', e \in \{\mathtt{W}, \mathtt{E}\}, e' \in \{\mathtt{R}, \mathtt{S}\}.\, t < t' \wedge \\ H(a)(t) = (-, e, \alpha) \wedge H(a')(t') = (-, e', \alpha') \end{array}\right\} \\
\mathtt{tar} & \triangleq & \left\{(\alpha, \alpha') \in \mathcal{T} \middle| \begin{array}{l} \exists a, a', t, t', e, e' \in \{\mathtt{W}, \mathtt{E}\}.\, t < t' \wedge \\ H(a)(t) = (-, e, \alpha) \wedge H(a')(t') = (-, e', \alpha') \end{array}\right\}
\end{array}$$

We will use notations $(\mathcal{T}, \mathsf{po}, \mathtt{vis}, \mathtt{tar}, H)$ to refer an element of the set:

$$\left\{(\mathcal{T}, \mathsf{po}, \mathtt{vis}, \mathtt{tar}, H) \middle| \begin{array}{l} (\mathcal{T}, \mathsf{po}) = \mathsf{programOrder}(\mathcal{L}, <) \wedge (\mathtt{vis}, \mathtt{tar}) = \mathsf{graph}(\mathcal{T}, H) \wedge \\ (\mathcal{L}, <, H, \eta) \in \bigcup_{n \in \mathbb{N}} \mathsf{traces}(H', \eta', n) \end{array}\right\}$$

For brevity, sometime we only mention and quantify few elements of this tuple but one should think they are quantified as an entire tuple.

Lemma A.7 (Separation)**.** If two transactions that are not associated by potentially arbitration relation $\mathtt{tar}$, they access different locations but commit at the same time.

$$\begin{array}{l}
\forall H, \mathtt{tar}, \alpha, \alpha'.\, (\alpha, \alpha'), (\alpha', \alpha) \notin \mathtt{tar} \\
\quad \implies \forall a, a', t, t', e, e \in \{\mathtt{W}, \mathtt{E}\}.\, H(a)(t) = (-, e, \alpha) \wedge H(a')(t') = (-, e', \alpha') \implies t = t' \wedge a \neq a'
\end{array}$$

Proof. Given the definition of $\mathtt{tar}$, $t = t'$ holds, and $a \neq a'$ is derived from Lemma A.3. ☐

Lemma A.8 (Semi-acyclic)**.** Both $\mathtt{vis}$ and $\mathtt{tar}$ are acyclic.

Proof. Proof by contradiction. Assume that there is a circle by $\mathtt{vis}$, which means that $\bigwedge_{0 \leq i \leq n} (\alpha_i, \alpha_{i+1}) \in \mathtt{vis} \wedge \alpha_0 = \alpha_n$ for some α_0 to α_n. By Lemma A.3, each transaction has a start time and a end time, thus let t_i^s and t_i^e be the start time and end time of the transaction α_i respectively. By Lemma A.2, we have $t_i^s < t_i^e$, and by definition of $\mathtt{vis}$, thus $t_i^e < t_{i+1}^s$. Therefore, $t_0^s < t_0^e < t_1^s < \cdots < t_n^s$, and we have contradiction that $t_0^s < t_n^s$.

Similarly for $\mathtt{tar}$, we have contradiction that $t_0^e < t_1^e < \cdots < t_n^e$. ☐

We can extend partial order $\texttt{tar}$ to a total order $\texttt{ar}$ by pick orders between those unrelated transactions. First, we define an auxiliary function that returns a set that includes the first transactions (might be more than two) that branch. To simplify, assume there is a unique initial transactions denoted by α_{init}, where $(\alpha_{init}, \alpha) \in \texttt{tar}$ for all α.

Definition A.9 (First Branch)**.**

$$
\begin{array}{rcl}
\mathsf{firstTrans}(\mathcal{T}, \texttt{tar}) & \triangleq & \alpha \texttt{ where } \forall \alpha' \in \mathcal{T}.\, \alpha = \alpha' \vee (\alpha, \alpha') \in \texttt{tar} \\
\mathsf{firstBranch}(\mathcal{T}, \texttt{tar}) & \triangleq & \begin{cases} \emptyset & \mathcal{T} = \emptyset \\ \mathsf{firstBranch}(\mathcal{T} \setminus \{\mathsf{firstTrans}(\mathcal{T}, \texttt{tar})\}, \texttt{tar}) & |\mathsf{dec}(\mathcal{T}, \texttt{tar})| = 1 \\ \mathsf{dec}(\mathcal{T}, \texttt{tar}) & |\mathsf{dec}(\mathcal{T}, \texttt{tar})| > 1 \end{cases} \\
\mathsf{dec}(\mathcal{T}, \texttt{tar}) & \triangleq & \{\alpha | \exists \alpha_{init} = \mathsf{firstTrans}(\mathcal{T}, \texttt{tar}).\, (\alpha_{init}, \alpha) \in \texttt{tar} \wedge \nexists \alpha'.\, (\alpha_{init}, \alpha'), (\alpha', \alpha) \in \texttt{tar}\}
\end{array}
$$

Intuitively, we take the first two transactions that are not connected, which definitely touch different locations by Lemma A.7, therefore we can simply pick a order and take the transitive closure. We continue the process until it is a total order.

Definition A.10 (Total relation)**.**

$$
\mathsf{toTotal}(\mathcal{T}, \texttt{tar}) \triangleq \begin{cases} (\mathcal{T}, \texttt{tar}) & \mathsf{firstBranch}(\mathcal{T}, \texttt{tar}) = \emptyset \\ \mathsf{toTotal}(\mathcal{T}, (\texttt{tar} \uplus \{(\alpha, \alpha')\})^{+}) & \alpha, \alpha' \in \mathsf{firstBranch}(\mathcal{T}, \texttt{tar}) \end{cases}
$$

We will use notations $(\mathcal{T}, \texttt{po}, \texttt{vis}, \texttt{ar}, H)$ to refer an element of the set:

$$
\{(\mathcal{T}, \texttt{po}, \texttt{vis}, \texttt{ar}, H) | (\mathcal{T}, \texttt{ar}) = \mathsf{toTotal}(\mathcal{T}, \texttt{tar})\}
$$

Lemma A.11 (Visibility)**.** Visibility relation should be a subset of arbitration relation, i.e. $\texttt{vis} \subseteq \texttt{ar}$.

Proof. For all transactions α and α', if $(\alpha, \alpha') \in \texttt{vis}$, it means that transaction α commits before α' starts, so that α must start before α' starts, by Lemma A.2. This means $(\alpha, \alpha') \in \texttt{tar}$ by the definition of $\texttt{tar}$, and because $\texttt{tar} \subseteq \texttt{ar}$, so $(\alpha, \alpha') \in \texttt{ar}$. □

Lemma A.12 (Monotonic time in a thread)**.** The thread's local time monotonically increase. This is

$$
\forall \sigma, \sigma', H, H', t, t', \mathbb{P}^{\uparrow}, \mathbb{P}^{\uparrow'}, \iota, a, t.\, (\sigma, H, t), \mathbb{P}^{\uparrow} \overset{\iota}{\rightsquigarrow}_{t} (\sigma', H', t'), \mathbb{P}^{\uparrow'} \implies t < t'
$$

Proof. Indication on the structure of the operational semantics. For base cases PChoiseLeft, PChoiseRight, PLoop, PSeqSkip and PPar, it is trivial since those rules do not change the time. For Commit, the premiss implies that $t' > t$ and for PWait, $t' = \max\{t, -\} \geq t$. For the inductive case PSeq, prove directly by the inductive hypothesis. □

Lemma A.13 (Monotonic time for fork and join)**.** The local times of two threads are greater then the time when the fork happens.

$$
\forall H, H', \eta, i, i', \sigma, \sigma', \mathbb{P}^{\uparrow}, \mathbb{P}^{\uparrow'}, t, t'.
$$

Proof. □

Lemma A.14 (Session)**.** $\texttt{po} \subseteq \texttt{vis}$.

Proof. By a case analysis of the definition of $\texttt{po}$. For α and α' where $\texttt{cmt}(i, \alpha) < \texttt{cmt}(i, \alpha')$, it means that transaction α is reduced by the semantics before α' is reduced. Then by Lemma A.12, the commit time of α is smaller than the start time of α', so that $(\alpha, \alpha') \in \texttt{vis}$.

For α and α' where $\texttt{cmt}(i, \alpha) < \texttt{fork}(i, i', -) < \texttt{cmt}(i', \alpha')$, first note that in the PFork the parent thread starts at time t and ends at t', and the child thread initialises with the time t', where in fact $t = t'$ By $\texttt{cmt}(i, \alpha) < \texttt{fork}(i, i', -)$, the transaction i is reduced before the fork, so by Lemma A.12, the end time of i is smaller than t. By $\texttt{fork}(i, i', -) < \texttt{cmt}(i', \alpha')$, the transaction i' is reduced after the fork and also by Lemma A.12, the start time of i is greater or equal to t'. Therefore, $(\alpha, \alpha') \in \texttt{vis}$.

Similarly, for α and α' where $\texttt{cmt}(i, \alpha) < \texttt{join}(i', i, -) < \texttt{cmt}(i', \alpha')$, by Lemma A.13, all the transactions by thread i must have smaller start and end times than all the transactions by thread i' after the join point. Therefore, $(\alpha, \alpha') \in \texttt{vis}$. □

SX: The session lemma I feel slightly unhappy, because it seems not very formal.

Lemma A.15 (Semi-prefix)**.** $\texttt{tar}; \texttt{vis} \subseteq \texttt{vis}$.

Proof. For all $\alpha, \alpha', \alpha''$, if $(\alpha, \alpha') \in \mathtt{tar}$ and $(\alpha', \alpha'') \in \mathtt{vis}$, by the definitions of $\mathtt{po}$ 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 $a, a'', t, t', t'', e \in \{\mathtt{W}, \mathtt{E}\}$ and $e'' \in \{\mathtt{R}, \mathtt{S}\}$ such that $H(a)(t) = (-, e, \alpha)$, $H(a'')(t'') = (-, e'', \alpha'')$ and $t < t''$, thus $(\alpha, \alpha'') \in \mathtt{vis}$. ☐

Lemma A.16 (Prefix)**.** $\mathtt{ar}; \mathtt{vis} \subseteq \mathtt{vis}$.

Proof. For all $\alpha, \alpha', \alpha''$ that $(\alpha, \alpha') \in \mathtt{ar}$ and $(\alpha', \alpha'') \in \mathtt{vis}$, if $(\alpha, \alpha') \in \mathtt{tar}$, it is proven by Lemma A.15. If $(\alpha, \alpha') \notin \mathtt{tar}$, it means it is a new edge by the $\mathsf{toTotal}$ function from Definition A.10. By the Lemma A.7, α and α' commit at the same time. By the definition of $\mathtt{vis}$, the commit time of α' is smaller than the start time of α''. Therefore, the commit time of α is also smaller than the start time of α'', so that $(\alpha, \alpha'') \in \mathtt{vis}$. ☐

Lemma A.17 (No conflict)**.** Two transactions cannot concurrently write to the same location, this means that one must observe another one. This is $\forall a, \alpha, \alpha'.\, H(a)(-) = (-, \mathtt{W}, \alpha) \wedge H(a)(-) = (-, \mathtt{W}, \alpha') \implies ((\alpha, \alpha') \in \mathtt{vis} \vee (\alpha', \alpha) \in \mathtt{vis}))$.

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 symmetry, 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 A.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 a, those two cases violate the $\mathtt{consist}$ requirement in the semantics, so one of the transactions must pick another start and end time. ☐

Lemma A.18 (External)**.** A transaction should read the last values it can observe. This means that for all transaction α and heap location a, if the transaction read a value v from the location, i.e. $\exists t.\, H(a)(t) = (v, \mathtt{R}, \alpha)$, 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 $\exists v', t'.\, H(a)(t') = (v', \mathtt{W}, \alpha') \implies v' = v$

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 $\forall \alpha'', t'', e''.\, H(a)(t'') = (-, e'', \alpha'') \implies e'' \neq \mathtt{W}$. Thus by the $\mathsf{startstate}$ in the semantics, we have $v' = v$. ☐

Lemma A.19 (Acyclic)**.** Both $\mathtt{vis}$ and $\mathtt{ar}$ are acyclic.

Proof. For $\mathtt{vis}$, it is proven by Lemma A.8.

To prove $\mathtt{ar}$ is acyclic, we prove inductively $\mathtt{tar}_i$ is acyclic, where $\mathtt{tar}_0 = \mathtt{tar}$, $\mathtt{tar}_n = \mathtt{ar}$ and $\mathtt{tar}_{i+1} = (\mathtt{tar}_i \uplus \{(\alpha, \alpha')\})^+$ for some α and α', which follows the process of $\mathsf{toTotal}$ from Definition A.10. For base case $\mathtt{tar}_0$, it is proven by Lemma A.8. Now assume $\mathtt{tar}_i$ is acyclic and $\mathtt{tar}_{i+1} = (\mathtt{tar}_i \uplus \{(\alpha, \alpha')\})^+$ for some α and α', we prove by contradiction that $\mathtt{tar}_{i+1}$ is acyclic. Given the hypothesis, if there is circle in $\mathtt{tar}_{i+1}$, such circle either contain the edge (α, α'), or can be broken down to a circle containing the edge (α, α'), because of the transitive closure. Note that by breaking down to a circle containing (α, α'), the rest edges inside the circle must be in $\mathtt{tar}_i$. Let assume the circle is $\alpha_0, \alpha_1, \ldots, \alpha_i, \alpha, \alpha', \alpha_{i+1}, \ldots, \alpha_n$ where $\alpha_0 = \alpha_n$. By the $\mathsf{toTotal}$ function from Definition A.10, $\alpha, \alpha' \in \mathsf{firstBranch}(\mathcal{T}, \mathtt{tar}_i)$, which means that for all α'', $(\alpha'', \alpha) \in \mathtt{tar}_i$ if and only if $(\alpha'', \alpha') \in \mathtt{tar}_i$, therefore $(\alpha'', \alpha) \in \mathtt{tar}_{i+1}$ if and only if $(\alpha'', \alpha') \in \mathtt{tar}_{i+1}$. This means that the sequence without α, i.e. $\alpha_0, \alpha_1, \ldots, \alpha_i, \alpha', \alpha_{i+1}, \ldots, \alpha_n$, is still a sequence where two adjacent transactions are connected by $\mathtt{tar}_{i+1}$, and this new sequence is a circle. Now all the edges in the new sequence/circle already exist in $\mathtt{tar}_i$, which violates our hypothesis. Therefore by contradiction, $\mathtt{tar}_{i+1}$ is acyclic. Given that, $\mathtt{ar}$ is acyclic. ☐

Lemma A.20 (Total order)**.** $\mathtt{ar}$ is a total order.

Proof. By Lemma A.19, $\mathtt{ar}$ is acyclic, and by Definition A.10, for all α and α', either $(\alpha, \alpha') \in \mathtt{ar}$ or $(\alpha', \alpha) \in \mathtt{ar}$. Therefore, it is a total order. ☐

Theorem A.21 (Soundness of the semantics)**.** The thread pool operational semantics $\rightsquigarrow_g$ (Def. 10) is sound.

Proof. By Lemma A.11, A.14, A.16, A.17, A.18, A.20. ☐