Leaf: Modularity for Temporary Sharing in Separation Logic

Technical Appendix

ANONYMOUS AUTHOR(S)

We include this appendix as a reference for additional technical content that we did not have space for in the main paper. We include:

- Reference for Leaf rules and important Iris rules (as present in the main body of the paper).
- Concrete counterexamples to explain why simpler versions of some Leaf rules would not be sound.
- A basic "counting logic" protocol.
- More details on the construction of the "hash table logic."
- A sketch of an expanded reader-writer lock with multiple reference counts.
- A sketch applying Leaf to heap semantics with non-atomic read and write operations.

Additional information can be found in the accompanying Coq formalization, which includes:

- Proofs for all the given Leaf laws
- Program logic for an atomic heap-based language
- The forever, fractional, and counting protocols
- Full case studies from the paper, including implementation, ghost state, and proofs:
  - The single-counter reader-writer lock
  - The linear-probing hash table
- Instantiation of the Iris adequacy theorem on the case studies

Contents

1 LEAF DEDUCTION RULES

1.1 Iris Background: PCM Custom Ghost State

Iris provides ghost state via an algebraic object called a *CMRA* (sometimes *camera*). A PCM is a special case of a CMRA, and in particular, a PCM is a special case of a *discrete* CMRA (i.e., a CMRA whose equality does not depend on the step-index). The version we present here is a simplified picture of Iris ghost state based on PCMs. Note that PCM-And relies on the discreteness so that $\preceq$ is well-defined.

Formally, a PCM is a set M (the *carrier* set) with a composition operator $\cdot : M \times M \to M$, and unit $\epsilon : M$, and validity predicate $\mathcal{V} : M \to \text{Bool}$, where we let,

$$a \preceq b \triangleq \exists c.\ a \cdot c = b$$

and,

$$\forall a, b.\ a \cdot b = b \cdot a$$
$$\forall a, b, c.\ a \cdot (b \cdot c) = (a \cdot b) \cdot c$$
$$\forall a.\ a \cdot \epsilon = a$$
$$\forall a, b.\ a \preceq b \land \mathcal{V}(b) \Rrightarrow \mathcal{V}(a)$$

Rules for PCM ghost state

Instantiated for a given PCM $(M, \cdot, \mathcal{V})$

Propositions: $\boxed{a}^{\gamma}$ (where $a : M,\ \gamma : \mathit{Name}$)

PCM-Unit
$$\text{True} \vdash \boxed{\epsilon}^{\gamma}$$

PCM-Valid
$$\boxed{a}^{\gamma} \vdash \mathcal{V}(a)$$

PCM-Sep
$$\boxed{a \cdot b}^{\gamma} \dashv\vdash \boxed{a}^{\gamma} * \boxed{b}^{\gamma}$$

PCM-Alloc
$$\frac{\mathcal{V}(a)}{\text{True} \Rrightarrow \exists \gamma.\ \boxed{a}^{\gamma}}$$

PCM-Update
$$\frac{a \rightsquigarrow b}{\boxed{a}^{\gamma} \Rrightarrow \boxed{b}^{\gamma}}$$

PCM-And
$$\frac{\forall t.\ ((x \preceq t) \land (y \preceq t) \land \mathcal{V}(t)) \Rrightarrow (z \preceq t)}{\boxed{x}^{\gamma} \land \boxed{y}^{\gamma} \vdash \boxed{z}^{\gamma}}$$

1.2 Elementary $\twoheadrightarrow$ deduction rules

Deduction rules for guarded resources

Persistent Propositions: $P \twoheadrightarrow_{\mathcal{E}} Q$ (where $P, Q : iProp$, $\mathcal{E} : \mathcal{P}(Name)$)

Guard-Refl
$$P \twoheadrightarrow_{\mathcal{E}} P$$

Guard-Trans
$$(P \twoheadrightarrow_{\mathcal{E}} Q) * (Q \twoheadrightarrow_{\mathcal{E}} R) \vdash (P \twoheadrightarrow_{\mathcal{E}} R)$$

Guard-Split
$$P * Q \twoheadrightarrow_{\mathcal{E}} P$$

Guard-Weaken-Mask
$$(G \twoheadrightarrow_{\mathcal{E}_1} P) \vdash (G \twoheadrightarrow_{\mathcal{E}_1 \cup \mathcal{E}_2} P)$$

Guard-Pers
$$\frac{\mathsf{persistent}(C)}{C \vdash (\mathsf{True} \twoheadrightarrow_{\mathcal{E}} C)}$$

Unguard-Pers
$$\frac{A * B \vdash C \quad \mathsf{persistent}(C)}{G * (G \twoheadrightarrow_{\mathcal{E}} A) * B \Rrightarrow_{\mathcal{E}} G * (G \twoheadrightarrow_{\mathcal{E}} A) * B * C}$$

Guard-Upd
$$\frac{\mathcal{E}_1 \cap \mathcal{E}_2 = \emptyset}{(G \twoheadrightarrow_{\mathcal{E}_1} P) * (P * A \Rrightarrow_{\mathcal{E}_2} P * B) \vdash (G * A \Rrightarrow_{\mathcal{E}_1 \cup \mathcal{E}_2} G * B)}$$

PointProp-Own
$$\mathsf{point}(\overset{\cdot}{x}{}^{\gamma})$$

Guard-Implies
$$\frac{A \vdash P \quad \mathsf{point}(P)}{(G \twoheadrightarrow_{\mathcal{E}} A) \vdash (G \twoheadrightarrow_{\mathcal{E}} P)}$$

Guard-And
$$\frac{A \wedge B \vdash P \quad \mathsf{point}(P)}{(G \twoheadrightarrow_{\mathcal{E}} A) * (G \twoheadrightarrow_{\mathcal{E}} B) \vdash (G \twoheadrightarrow_{\mathcal{E}} P)}$$

PointProp-Sep
$$\frac{\mathsf{point}(P) \quad \mathsf{point}(Q)}{\mathsf{point}(P * Q)}$$

1.3 Storage Protocols

A **storage protocol** consists of:
A *storage monoid*, that is, a partial commutative monoid $(S, \cdot, \mathcal{V})$, where,

$$\forall a.\ a \cdot \epsilon = a$$
$$\forall a, b.\ a \cdot b = b \cdot a$$
$$\forall a, b, c.\ (a \cdot b) \cdot c = a \cdot (b \cdot c)$$
$$\mathcal{V}(\epsilon)$$
$$\forall a, b.\ a \preceq b \land \mathcal{V}(b) \Rightarrow \mathcal{V}(a)$$

A *protocol monoid*, that is, a (total) commutative monoid $(P, \cdot)$, with an arbitrary predicate $C : P \to \text{Bool}$ and function $\mathcal{S} : P|_C \to S$ (i.e., the domain of $\mathcal{S}$ is restricted to the subset of P where C holds) where,

$$\forall a.\ a \cdot \epsilon = a$$
$$\forall a, b.\ a \cdot b = b \cdot a$$
$$\forall a, b, c.\ (a \cdot b) \cdot c = a \cdot (b \cdot c)$$
$$\forall a.\ C(a) \Rightarrow \mathcal{V}(\mathcal{S}(a))$$

Note that C (unlike $\mathcal{V}$) is *not* necessarily closed under $\preceq$.
Additional notation for storage protocols:

For $p, p' : P$ and $s, s' : S$, define:

$$\begin{aligned}
(p, s) \dashrightarrow (p', s') &\triangleq \forall q.\ C(p \cdot q) \Rightarrow (C(p' \cdot q) \\
&\quad \land \mathcal{V}(\mathcal{S}(p \cdot q) \cdot s) \\
&\quad \land \mathcal{S}(p \cdot q) \cdot s = \mathcal{S}(p' \cdot q) \cdot s') \\
p \dashrightarrow (p', s') &\triangleq (p, \epsilon) \dashrightarrow (p', s') \\
(p, s) \dashrightarrow p' &\triangleq (p, s) \dashrightarrow (p', \epsilon) \\
p \rightsquigarrow p' &\triangleq (p, \epsilon) \dashrightarrow (p', \epsilon) \\
p \twoheadrightarrow s &\triangleq \forall q.\ C(p \cdot q) \Rightarrow s \preceq \mathcal{S}(p \cdot q)
\end{aligned}$$

(a) Definition of a storage protocol and its derived relations.

Storage Protocol Logic
Instantiated for a given storage protocol $(S, \cdot, \mathcal{V}), (P, \cdot), C, \mathcal{S}$
Propositions: $\langle p \rangle^\gamma$
Persistent propositions: $\text{sto}(\gamma, F)$
(where $\gamma : \mathit{Name},\ F : S \to \mathit{iProp},\ p : P$)

$\text{RespectsComposition}(F) \triangleq (F(\epsilon) \dashv\vdash \text{True})$
and $\forall x, y.\ \mathcal{V}(x \cdot y) \Rightarrow (F(x \cdot y) \dashv\vdash F(x) * F(y))$

SP-Alloc
$$\frac{\text{RespectsComposition}(F) \quad C(p) \quad \mathcal{N}\ \text{infinite}}{F(\mathcal{S}(p)) \Rrightarrow \exists \gamma.\ \text{sto}(\gamma, F) * \langle p \rangle^\gamma * (\gamma \in \mathcal{N})}$$

SP-Exchange
$$\frac{(p, s) \dashrightarrow (p', s')}{\text{sto}(\gamma, F) \vdash (\triangleright F(s)) * \langle p \rangle^\gamma \Rrightarrow_\gamma (\triangleright F(s')) * \langle p' \rangle^\gamma}$$

SP-Deposit
$$\frac{(p, s) \dashrightarrow p'}{\text{sto}(\gamma, F) \vdash (\triangleright F(s)) * \langle p \rangle^\gamma \Rrightarrow_\gamma \langle p' \rangle^\gamma}$$

SP-Withdraw
$$\frac{p \dashrightarrow (p', s')}{\text{sto}(\gamma, F) \vdash \langle p \rangle^\gamma \Rrightarrow_\gamma (\triangleright F(s')) * \langle p' \rangle^\gamma}$$

SP-Update
$$\frac{p \rightsquigarrow p'}{\text{sto}(\gamma, F) \vdash \langle p \rangle^\gamma \Rrightarrow_\gamma \langle p' \rangle^\gamma}$$

SP-PointProp
$\text{point}(\langle p \rangle^\gamma)$

SP-Guard
$$\frac{p \twoheadrightarrow s}{\text{sto}(\gamma, F) \vdash \langle p \rangle^\gamma \twoheadrightarrow_\gamma (\triangleright F(s))}$$

SP-Unit
$\text{sto}(\gamma, F) \vdash \langle \epsilon \rangle^\gamma$

SP-Sep
$\langle p \rangle^\gamma * \langle q \rangle^\gamma \dashv\vdash \langle p \cdot q \rangle^\gamma$

SP-Valid
$\langle p \rangle^\gamma \vdash \exists q.\ C(p \cdot q)$

(b) Deduction rules relating to a storage protocol and derived relations.

1.4 Remarks and Counterexamples

We provide counterexamples to rules that one might initially expect, or hope, to be true.

1.4.1 The restriction on Guard-Implies. Suppose this rule held:

$$\textsc{Wrong-Guard-Implies}\ \frac{A \vdash P}{(G \twoheadrightarrow_{\mathcal{E}} A) \vdash (G \twoheadrightarrow_{\mathcal{E}} P)}$$

We will derive a contradiction. Take any propositions P, Q such that

$$\mathsf{True} \Rrightarrow P * P \qquad P * P \Rrightarrow Q * Q \qquad P * Q \Rrightarrow \mathsf{False}$$

Then we have,

$$\begin{aligned} P * (P \vee Q) &\Rrightarrow P * P \\ &\Rrightarrow Q * Q \\ &\Rrightarrow Q * (P \vee Q) \end{aligned}$$

Now, $P \vdash (P \vee Q)$ and so $P \twoheadrightarrow (P \vee Q)$ and thus by Guard-Upd,

$$P * P \Rrightarrow P * Q$$

which lets us derive a contradiction,

$$\begin{aligned} \mathsf{True} &\Rrightarrow P * P \\ &\Rrightarrow P * Q \\ &\Rrightarrow \mathsf{False} \end{aligned}$$

1.4.2 The restriction on Guard-And. Suppose this rule held:

$$\textsc{Wrong-Guard-And}\ \frac{A \wedge B \vdash P}{(G \twoheadrightarrow_{\mathcal{E}} A) * (G \twoheadrightarrow_{\mathcal{E}} B) \vdash (G \twoheadrightarrow_{\mathcal{E}} P)}$$

We will derive a contradiction. Take any propositions A, B, C such that,

$$\mathsf{True} \Rrightarrow A * B * C \qquad A * A \vdash \mathsf{False} \qquad B * B \vdash \mathsf{False} \qquad C * C \vdash \mathsf{False} \qquad B \wedge C \vdash B * C$$

By Guard-Split we have,

$$\begin{aligned} (A \vee B) * (A \vee C) &\twoheadrightarrow (A \vee B) \\ (A \vee B) * (A \vee C) &\twoheadrightarrow (A \vee C) \end{aligned}$$

So by Wrong-Guard-And, we have,

$$(A \vee B) * (A \vee C) \twoheadrightarrow (A \vee B) \wedge (A \vee C)$$

Then we have,

$$\begin{aligned} A * ((A \vee B) \wedge (A \vee C)) &\vdash A * (B \wedge C) \\ &\vdash (B \wedge C) * A \\ &\vdash (B \wedge C) * ((A \vee B) \wedge (A \vee C)) \end{aligned}$$

By Guard-Upd, we have

$$A * ((A \vee B) * (A \vee C)) \vdash (B \wedge C) * ((A \vee B) * (A \vee C))$$

Finally,

$$\begin{aligned}\mathsf{True} &\Rrightarrow A * B * C\\ &\Rrightarrow A * ((A \lor B) * (A \lor C))\\ &\Rrightarrow (B \land C) * ((A \lor B) * (A \lor C))\\ &\Rrightarrow (B * C) * ((A \lor B) * (A \lor C))\\ &\Rrightarrow \mathsf{False}\end{aligned}$$

1.4.3 The restriction on Guard-Upd. Suppose this rule held:

$$\textsc{Wrong-Guard-Upd}\quad (G \twoheadrightarrow_{\mathcal{E}_1} P) * (P * A \Rrightarrow_{\mathcal{E}_2} P * B) \vdash (G * A \Rrightarrow_{\mathcal{E}_1 \cup \mathcal{E}_2} G * B)$$

We will derive a contradiction.

Take fractional memory permissions. We have $(\ell \xhookrightarrow{\mathsf{frac}}_{1/2} v) \twoheadrightarrow_{\mathit{Frac}} (\ell \hookrightarrow v)$. We also have,

$$(\ell \hookrightarrow v) * (\ell \hookrightarrow v) \vdash \mathsf{False}$$

Therefore,

$$(\ell \hookrightarrow v) * (\ell \hookrightarrow v) \Rrightarrow \mathsf{False} * (\ell \hookrightarrow v) * (\ell \hookrightarrow v)$$

We can apply Wrong-Guard-Upd twice:

$$\begin{aligned}(\ell \xhookrightarrow{\mathsf{frac}}_{1/2} v) * (\ell \hookrightarrow v) &\Rrightarrow_{\mathit{Frac}} \mathsf{False} * (\ell \xhookrightarrow{\mathsf{frac}}_{1/2} v) * (\ell \hookrightarrow v)\\ (\ell \xhookrightarrow{\mathsf{frac}}_{1/2} v) * (\ell \xhookrightarrow{\mathsf{frac}}_{1/2} v) &\Rrightarrow_{\mathit{Frac}} \mathsf{False} * (\ell \xhookrightarrow{\mathsf{frac}}_{1/2} v) * (\ell \xhookrightarrow{\mathsf{frac}}_{1/2} v)\end{aligned}$$

And so:

$$(\ell \xhookrightarrow{\mathsf{frac}}_{1} v) \Rrightarrow_{\mathit{Frac}} \mathsf{False}$$

from which we can derive a contradiction.

1.4.4 The lack of any $*$ rule. We might expect a rule like,

$$\textsc{Wrong-Guard-Sep-Disjoint}\quad \frac{\mathcal{E}_1 \cap \mathcal{E}_2 = \emptyset}{(G_1 \twoheadrightarrow_{\mathcal{E}_1} P_1) * (G_1 \twoheadrightarrow_{\mathcal{E}_2} P_2) \vdash (G_1 * G_2) \twoheadrightarrow_{\mathcal{E}_1 \cup \mathcal{E}_2} (P_1 * P_2)}$$

A special case would be,

$$(G \twoheadrightarrow_{\mathcal{E}} P) \vdash (G * A \twoheadrightarrow_{\mathcal{E}} P * A)$$

In fact, not even the special case is sound. To see why requires us to put an element of a storage protocol inside itself. At a very high level: take any timeless proposition P such that $P * P \vdash \mathsf{False}$. Insert P into a storage protocol and obtain some Q such that $Q \twoheadrightarrow_{\mathcal{E}} P$. Then insert Q into the same protocol and obtain R such that $R \twoheadrightarrow_{\mathcal{E}} (P * Q)$. Now, if the above rule held, we could get $(P * Q) \twoheadrightarrow_{\mathcal{E}} (P * P)$. Then by Guard-Refl we could get $R \twoheadrightarrow_{\mathcal{E}} P * P$, which should be impossible.

Let us demonstrate in a little more detail that this argument really is possible.

It is not hard to construct a protocol, for any $X : \mathit{Set}$ and $F : X \to \mathit{iProp}$, such that, upon initialization, we have the rules,

$$\begin{aligned}F(x) &\Rrightarrow_{\{\gamma\}} g(\gamma, x)\\ g(\gamma, x) &\twoheadrightarrow_{\{\gamma\}} \triangleright F(x)\\ g(\gamma, x_1) * g(\gamma, x_2) &\twoheadrightarrow_{\{\gamma\}} (\triangleright F(x_1)) * (\triangleright F(x_2))\end{aligned}$$

where $g(\gamma, x)$ are duplicable propositions. Let $X = \{\omega\} \cup \mathit{Name}$ for some special value ω. Note that this is enough to define the storage protocol.

Therefore we can now define F. Now set:

$$F(\omega) \triangleq P \qquad F(\gamma) \triangleq g(\gamma, \omega)$$

We can now deposit $F(\omega) = P$ to get $g(\gamma, \omega)$. Call this new proposition Q. We then have $Q \twoheadrightarrow\!\!\!\!\twoheadrightarrow_{\{\mathcal{E}\}} P$.

Observe that $Q = g(\gamma, \omega) = F(\gamma)$. We can thus deposit Q as well and get $g(\gamma, \gamma)$.

Finally, let $R = g(\gamma, \gamma) * g(\gamma, \omega)$. We then have $R \twoheadrightarrow\!\!\!\!\twoheadrightarrow_{\{\gamma\}} P * Q$. This completes the outline above.

2 DERIVATION OF A COUNTING PROTOCOL

We show how to build a counting protocol, an analogue of Example 3.2 from the main paper, which does a fractional protocol.

Specifically, let us suppose we have a single proposition, Q, we would like to manage. We are not assuming that $Q * Q \vdash \mathsf{False}$, which means that we might be able to have multiple instances of Q managed by the protocol at once, with multiple counters.

We want to construct a commutative monoid spanned by some elements ref and $\mathsf{counter}(n)$ modulo the identity $\mathsf{ref} \cdot \mathsf{counter}(n) = \mathsf{counter}(n-1)$. An easy way to do this is to take the monoid,

$$P \triangleq \{(r, c) : \mathbb{Z} \times \mathbb{N} \mid c = 0 \Rightarrow r \leq 0\}$$

We define composition to simply be pairwise addition. Let,

$$\mathsf{ref} \triangleq (-1, 0)$$
$$\mathsf{counter}(r) \triangleq (r, 1)$$

We want the counts from the ref objects to "cancel out" the counters' counts, so we set:

$$C((r, c)) \triangleq (r = 0)$$

Finally, take the storage protocol to be $S \triangleq \mathbb{N}$, and the storage function to be

$$\mathcal{S}((r, c)) \triangleq c$$

One might notice in this set-up it is possible for counters to be negative, which seems a little odd, but it does not matter since any negative counter would be canceled out by some other positive counter.

At any rate, we have,

$$\mathsf{ref} \cdot \mathsf{counter}(r) = \mathsf{counter}(r-1)$$
$$(\epsilon, 1) \rightsquigarrow (\mathsf{counter}(0), 0)$$
$$(\mathsf{counter}(0), 0) \rightsquigarrow (\epsilon, 1)$$
$$\mathsf{ref} \twoheadrightarrow 1$$

We could write these as:

$$(-1, 0) \cdot (r, 1) = (r-1, 1)$$
$$((0, 0), 1) \rightsquigarrow ((0, 1), 0)$$
$$((0, 1), 0) \rightsquigarrow ((0, 0), 1)$$
$$(-1, 0) \twoheadrightarrow 1$$

For the last one, it is crucial that we cannot write $(-1, 0) \cdot (1, 0) = (0, 0)$ because $(1, 0)$ is disallowed by the condition in the definition of P. Observe that we had to include this condition in the definition of P (rather than, say, imposing it via C) because the condition is not closed under $\preceq$.

As such, if we have $(-1, 0) \cdot (r, c) = (0, c)$ then we must have $c > 0$, i.e., $c \geq 1$. Hence $(-1, 0) \twoheadrightarrow 1$.

3 LANGUAGE SYNTAX AND PROGRAM LOGIC

In our case studies, we use a heap-based language with values and expressions given by:

$$v := \mathsf{True} \mid \mathsf{False} \mid n \mid \texttt{rec}\ f(x).\ e \mid () \mid (v_1, v_2) \mid \texttt{inl}(v) \mid \texttt{inr}(v) \mid \ell$$

$$e := v \mid x \mid e_1 e_2 \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid e_1; e_2 \mid (e_1, e_2) \mid \texttt{inl}(e) \mid \texttt{inr}(e) \mid \pi_i(e) \mid (\texttt{match}\ e\ \texttt{with}\ \ldots)$$
$$\mid (\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) \mid \texttt{fork}\ e \mid e_1 + e_2 \mid e_1 = e_2$$
$$\mid \texttt{abort}$$
$$\mid \texttt{ref}(e) \mid \texttt{free}(e) \mid !e \mid e_1 \leftarrow e_2 \mid \texttt{CAS}(e_1, e_2, e_3) \mid \texttt{FetchAdd}(e_1, e_2)$$

Higher level constructs like records, arrays, or `do ... until` can be considered as syntactic sugar around a straightforward encoding.

We use standard operational semantics over a heap state $\sigma : \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Value}$. In particular, we use the following head reduction steps for the heap operations:

$$\begin{aligned}
(\texttt{ref}(v), \sigma) &\to (\ell, \sigma[\ell \mapsto v]) && \text{where } \ell \notin \sigma \\
(\texttt{free}(\ell), \sigma) &\to ((), \sigma \backslash \{\ell\}) && \text{where } \ell \in \sigma \\
(!\ell, \sigma) &\to (v, \sigma) && \text{where } \ell \in \sigma \text{ and } \sigma(\ell) = v \\
(\ell \leftarrow v, \sigma) &\to ((), \sigma[\ell \mapsto v]) && \text{where } \ell \in \sigma \\
(\texttt{CAS}(\ell, v_0, v_1), \sigma) &\to (\mathsf{True}, \sigma[\ell \mapsto v_1]) && \text{where } \ell \in \sigma \text{ and } \sigma(\ell) = v_0 \\
(\texttt{CAS}(\ell, v_0, v_1), \sigma) &\to (\mathsf{False}, \sigma) && \text{where } \ell \in \sigma \text{ and } \sigma(\ell) \neq v_0 \\
(\texttt{FetchAdd}(\ell, n), \sigma) &\to (m, \sigma[\ell \mapsto n + m]) && \text{where } \ell \in \sigma \text{ and } \sigma(\ell) = m
\end{aligned}$$

Within Iris, using its standard method of defining the weakest-precondition and proving an adequacy theorem, we can prove the following heap rules:

HEAP-REF
$\{\}\ \texttt{ref}(v)\ \{\ell.\ \ell \hookrightarrow v\}$

HEAP-FREE
$\{\ell \hookrightarrow v\}\ \texttt{free}(v)\ \{\}$

HEAP-WRITE
$\{\ell \hookrightarrow v\}\ !\ell\ \{r.\ \ell \hookrightarrow v * v = r\}$

HEAP-READ
$\{\ell \hookrightarrow v\}\ \ell \leftarrow v'\ \{\ell \hookrightarrow v'\}$

HEAP-FETCHADD
$\{\ell \hookrightarrow n\}\ \texttt{FetchAdd}(\ell, m)\ \{v.\ (v = m) * \ell \hookrightarrow (n + m)\}$

HEAP-CAS-TRUE
$\{\ell \hookrightarrow v\}\ \texttt{CAS}(\ell, v, v')\ \{r.\ (r = \mathsf{True}) * \ell \hookrightarrow v'\}$

HEAP-CAS-FALSE
$\{\ell \hookrightarrow v * (v \neq v_1)\}\ \texttt{CAS}(\ell, v_1, v')\ \{r.\ (r = \mathsf{False}) * \ell \hookrightarrow v\}$

We further define the shorthand:

$$[X]_{\mathcal{E}}\ [\cdots]\ \{P\}\ e\ \{Q\} \triangleq \forall G.\ [\cdots]\ \{P * G * (G \twoheadrightarrow_{\mathcal{E}} X)\}\ e\ \{Q * G\}$$

And can show:

HEAP-READ-SHARED
$[\ell \hookrightarrow v]\ \{\}\ !\ell\ \{v'.\ v = v'\}$

4 EXPANDED READER-WRITER LOCK: MULTIPLE REFERENCE COUNTERS

The case study from the paper and the Coq formalization uses a single reference counter. Here, we sketch (on paper only) an expanded version for multiple reference counters. The multiple reference counter version can be useful to reduce thread contention in reader-heavy workloads.

Specifically, we consider the following implementation, for a fixed value K, the number of counters. To acquire a read lock, the client supplies an integer $k \in [0, K)$ explaining which counter they will use. To acquire an exclusive lock, the client has to check all counters.

```
multi_rwlock_new() ≜
  let rcs = for k in [0, K) do
    ref(0)
  done
  {exc : ref(False), rcs : rcs}

multi_rwlock_free(rw) ≜
  free(rw.exc);
  for k in [0, K) do
    free(rw.rcs[k])
  done
free(rw.rcs)

multi_lock_exc(rw) ≜
  do
    let success = CAS(rw.exc, False, True)
  until success
  for k in [0, K) do
    do
      let r = !rw.rcs
    until r = 0
  done

multi_unlock_exc(rw) ≜
  rw.exc ← 0

multi_lock_shared(rw, k) ≜
  do
    FetchAdd(rw.rcs[k], 1);
    let exc = !rw.exc in
    if exc then FetchAdd(rw.rcs[k], −1);
  until exc = False

multi_unlock_shared(rw, k) ≜
  FetchAdd(rw.rcs[k], −1)
```

Our specification is mostly unchanged; we just need to account for k in the shared locks.

Multi-counter RwLock Specification

Propositions: $\mathsf{IsRwLock}(rw, \gamma, F)$ $\mathsf{Exc}(\gamma)$ $\mathsf{Sh}(\gamma, k, x)$

(where $rw : \mathit{Value}$, $\gamma : \mathit{Name}$, $X : \mathit{Set}$, $x : X$, $F : X \to \mathit{iProp}$, $k \in \mathbb{N}$ and $0 \le k < K$)

$$\begin{array}{rrll}
\forall F, x. & & \{F(x)\}\ \mathsf{multi_rwlock_new}() & \{rw.\ \exists \gamma, \mathsf{IsRwLock}(rw, \gamma, F)\} \\
\forall rw, \gamma, F. & \{\mathsf{IsRwLock}(rw, \gamma, F)\} & \mathsf{multi_rwlock_free}(rw) & \{\} \\
\forall rw, \gamma, F. & [\mathsf{IsRwLock}(rw, \gamma, F)] & \{\}\ \mathsf{multi_lock_exc}(rw) & \{\mathsf{Exc}(\gamma) * \exists x.\ \triangleright F(x)\} \\
\forall rw, \gamma, F, x. & [\mathsf{IsRwLock}(rw, \gamma, F)] & \{\mathsf{Exc}(\gamma) * F(x)\}\ \mathsf{multi_unlock_exc}(rw) & \{\} \\
\forall rw, \gamma, F. & [\mathsf{IsRwLock}(rw, \gamma, F)] & \{\}\ \mathsf{multi_lock_shared}(rw, k) & \{\exists x.\ \mathsf{Sh}(\gamma, k, x)\} \\
\forall rw, \gamma, F, x. & [\mathsf{IsRwLock}(rw, \gamma, F)] & \{\mathsf{Sh}(\gamma, k, x)\}\ \mathsf{multi_unlock_shared}(rw, k) & \{\}
\end{array}$$

$$\forall rw, \gamma, F, x.\ \mathsf{IsRwLock}(rw, \gamma, F) \vdash (\mathsf{Sh}(\gamma, k, x) \twoheadrightarrow \triangleright F(x))$$

To verify the implementation against the spec, it suffices to construct the following reader-writer lock logic. Again, this is a variant on the one from the simpler lock. The main difference is that ShPending and Sh both take an additional parameter k (the index of the counter being incremented) and ExcPending takes an additional parameter j (the number of counters that have been checked so far). We have one rule (Rw-Exc-Begin) to enter the "pending" phase, one rule (Rw-Exc-Progress) to advance the counter j, and one rule (Rw-Exc-Acquire) to take the lock and perform the withdraw.

RwLock Logic

Propositions: $\mathsf{Fields}(\gamma, \mathit{exc}, \mathit{rcs}, x)$ $\mathsf{ExcPending}(\gamma, j)$ $\mathsf{Exc}(\gamma)$ $\mathsf{ShPending}(\gamma, k)$ $\mathsf{Sh}(\gamma, k, x)$

Persistent Propositions: $\mathsf{RwFamily}(\gamma, F)$

(where $\gamma : \mathit{Name}$, $\mathit{exc} : \mathsf{Bool}$, $\mathit{rcs} : \mathbb{Z}^K$, $X : \mathit{Set}$, $x : X$, $F : X \to \mathit{iProp}$, $j, k : \mathbb{N}$ and $0 \le k < K$ and $0 \le j \le K$)

$$F(x) \Rrightarrow \exists \gamma.\ \mathsf{Fields}(\gamma, \mathsf{False}, 0, x) * \mathsf{RwFamily}(\gamma, F) \qquad \text{(Rw-Init)}$$

$$\mathsf{RwFamily}(\gamma, F) \vdash \mathsf{Fields}(\gamma, \mathsf{False}, \mathit{rcs}, x) \Rrightarrow_\gamma \mathsf{Fields}(\gamma, \mathsf{True}, \mathit{rcs}, x) * \mathsf{ExcPending}(\gamma, 0) \qquad \text{(Rw-Exc-Begin)}$$

$$(j < K \land \mathit{rcs}(j) = 0) * \mathsf{Fields}(\gamma, \mathit{exc}, \mathit{rcs}, x) * \mathsf{ExcPending}(\gamma, j) \Rrightarrow_\gamma \mathsf{Fields}(\gamma, \mathit{exc}, \mathit{rcs}, x) * \mathsf{ExcPending}(\gamma, j+1) \qquad \text{(Rw-Exc-Progress)}$$

$$\mathsf{Fields}(\gamma, \mathit{exc}, \mathit{rcs}, x) * \mathsf{ExcPending}(\gamma, K) \Rrightarrow_\gamma \mathsf{Fields}(\gamma, \mathit{exc}, \mathit{rcs}, x) * \mathsf{Exc}(\gamma) * \triangleright F(x) \qquad \text{(Rw-Exc-Acquire)}$$

$$\mathsf{Fields}(\gamma, \mathit{exc}, \mathit{rcs}, y) * \mathsf{Exc}(\gamma) * \triangleright F(x) \Rrightarrow_\gamma \mathsf{Fields}(\gamma, \mathsf{False}, \mathit{rcs}, x) \qquad \text{(Rw-Exc-Release)}$$

$$\mathsf{Fields}(\gamma, \mathit{exc}, \mathit{rcs}, x) \Rrightarrow_\gamma \mathsf{Fields}(\gamma, \mathit{exc}, \mathit{rcs}[k \mapsto \mathit{rcs}(k)+1], x) * \mathsf{ShPending}(\gamma, k) \qquad \text{(Rw-Shared-Begin)}$$

$$\mathsf{Fields}(\gamma, \mathsf{False}, \mathit{rcs}, x) * \mathsf{ShPending}(\gamma, k) \Rrightarrow_\gamma \mathsf{Fields}(\gamma, \mathsf{False}, \mathit{rcs}, x) * \mathsf{Sh}(\gamma, k, x) \qquad \text{(Rw-Shared-Acquire)}$$

$$\mathsf{Fields}(\gamma, \mathit{exc}, \mathit{rcs}, x) * \mathsf{Sh}(\gamma, k, y) \Rrightarrow_\gamma \mathsf{Fields}(\gamma, \mathit{exc}, \mathit{rcs}[k \mapsto \mathit{rcs}(k)-1], x) \qquad \text{(Rw-Shared-Release)}$$

$$\mathsf{Fields}(\gamma, \mathit{exc}, \mathit{rcs}, x) * \mathsf{ShPending}(\gamma, k) \Rrightarrow_\gamma \mathsf{Fields}(\gamma, \mathit{exc}, \mathit{rcs}[k \mapsto \mathit{rcs}(k)-1], x) \qquad \text{(Rw-Shared-Retry)}$$

$$\mathsf{Sh}(\gamma, k, x) \twoheadrightarrow_\gamma \triangleright F(x) \qquad \text{(Rw-Shared-Guard)}$$

Once again, we take the storage monoid S to be $\textsc{Excl}(X)$. For the storage protocol construction, we first pick a monoid for each element:

- For Fields, we use $\textsc{Excl}(\text{Bool} \times \mathbb{Z}^K \times X)$
- For ExcPending, we use $\textsc{Excl}([0, K])$
- For Exc, we use $\textsc{Excl}(\mathbf{1})$
- For ShPending, we use $\mathbb{N}^K$
- For Sh, we use a variant on $\textsc{AgN}(X)$ from the simpler version. Specifically, we define $\textsc{AgNVec}(X)$ which uses a vector of counts instead of a single integer. Specifically, $\textsc{AgNVec}(X)$ has elements,

$$\epsilon \mid \text{agn}(x, n) \mid \lightning$$

where $n \in \mathbb{N}^k$ and n is nonzero. Once again, we let $\text{agn}(x, n) \cdot \text{agn}(x, m) = \text{agn}(x, n + m)$, where $n + m$ is elementwise vector addition, and $\text{agn}(x, n) \cdot \text{agn}(y, m) = \lightning$ for $x \neq y$.

All in all, we take our protocol monoid P to be the product,

$$\textsc{Excl}(\text{Bool} \times \mathbb{Z}^K \times X) \times \textsc{Excl}([0, K]) \times \textsc{Excl}(\mathbf{1}) \times \mathbb{N}^K \times \textsc{AgNVec}(X)$$

Now we let,

$$\begin{array}{rlllll}
\text{fields}(exc, rcs, x) & \triangleq (\text{ex}((exc, rcs, x)), & \epsilon, & \epsilon, & 0, & \epsilon) \\
\text{excPending}(j) & \triangleq (\epsilon, & \text{ex}(j), & \epsilon, & 0, & \epsilon) \\
\text{exc} & \triangleq (\epsilon, & \epsilon, & \text{ex}, & 0, & \epsilon) \\
\text{shPending}(k) & \triangleq (\epsilon, & \epsilon, & \epsilon, & [k \mapsto 1], & \epsilon) \\
\text{sh}(k) & \triangleq (\epsilon, & \epsilon, & \epsilon, & 0, & \text{agn}(x, [k \mapsto 1]))
\end{array}$$

Let $\text{Fields}(\gamma, exc, rcs, x) \triangleq \langle \text{fields}(exc, rcs, x) \rangle^\gamma$ and so on, and also let $\text{RwFamily}(\gamma, F) \triangleq \text{sto}(\gamma, F)$.

We define $\mathcal{S}$:

$$\mathcal{S}(\text{ex}((exc, rcs, x)), \_, \epsilon, \_, \_) \triangleq \text{ex}(x)$$
$$\mathcal{S}(\text{ex}((exc, rcs, x)), \_, \text{ex}, \_, \_) \triangleq \epsilon$$

We define C to be False if any entry is $\lightning$ or the first entry is ϵ; otherwise,

$$C((\text{ex}((exc, rcs, x)), ep, e, sp, s)) \triangleq (\forall k.\ rcs(k) = sp(k) + count(k, s)) \wedge (\neg exc \Rightarrow ep = \epsilon \wedge e = \epsilon)$$
$$\wedge (exc \Rightarrow (ep \neq \epsilon \vee e \neq \epsilon) \wedge \neg(ep \neq \epsilon \wedge e \neq \epsilon)) \wedge (e \Rightarrow s = \epsilon) \wedge (\forall y, n.\ s = \text{agn}(y, n) \Rightarrow x = y)$$
$$\wedge (\forall k, k'.\ ep = \text{ex}(k') \wedge k < k' \Rightarrow count(k, s) = 0)$$

(where $count(k, \text{agn}(x, n)) = n(k)$ and $count(k, \epsilon) = 0$)

Now, we can show:

- Rw-Exc-Acquire by SP-Withdraw
- Rw-Exc-Release by SP-Deposit
- Rw-Shared-Guard by SP-Guard
- The rest via SP-Update

5 HASH TABLE CUSTOM GHOST STATE CONSTRUCTION

Fix an integer $L > 0$ and a hash function $H : Key \to \mathbb{N}$. Consider the monoid

$$(Key \rightharpoonup \text{Excl}(Value^?)) \times (\mathbb{N} \rightharpoonup \text{Excl}((Key \times Value)^?)).$$

Define validity by:

$$\begin{aligned}
\text{KeysDistinct}((\sigma,\mu)) &\triangleq \forall i_1, k_1, v_1, i_2, v_2.\ \sigma(i_1) = \text{ex}((k_1,v_1)) \land \sigma(i_2) = \text{ex}((k_2,v_2)) \land i_1 \neq i_2 \Rightarrow k_1 \neq k_2 \\
\text{MapConsistent}((\sigma,\mu)) &\triangleq \forall k, v.\ \mu(k) = \text{ex}(\text{Some}(v)) \Rightarrow \exists i.\ \sigma(i) = \text{ex}(\text{Some}((k,v))) \\
\text{SlotsConsistent}((\sigma,\mu)) &\triangleq \forall i, k, v.\ \sigma(i) = \text{ex}(\text{Some}((k,v))) \Rightarrow k \in \mu \land \mu(k) \neq \text{ex}(\text{None}) \\
\text{Contiguous}((\sigma,\mu)) &\triangleq \forall i, k, v.\ \mu(i) = \text{ex}(\text{Some}((k,v))) \Rightarrow H(k) \leq i \\
&\quad \land (\forall j.\ H(k) \leq j \leq i \Rightarrow j \in \mu \land \mu(j) \neq \text{ex}(\text{None}))
\end{aligned}$$

$$\begin{aligned}
P((\sigma,\mu)) &\triangleq (\forall i.\ i \in \sigma \Rightarrow \exists s.\ \sigma(i) = \text{ex}(s)) \\
&\quad \land (\forall k.\ k \in \mu \Rightarrow \exists v.\ \mu(k) = \text{ex}(v)) \\
&\quad \land \text{KeysDistinct}((\sigma,\mu)) \\
&\quad \land \text{MapConsistent}((\sigma,\mu)) \\
&\quad \land \text{SlotsConsistent}((\sigma,\mu)) \\
&\quad \land \text{Contiguous}((\sigma,\mu)) \\
\mathcal{V}(z) &\triangleq \exists z'.\ z \preceq z' \land P(z')
\end{aligned}$$

These invariants can be interpreted as follows:

- KeysDistinct says all slots in the hash table have distinct keys.
- MapConsistent says that for each key-value pair (k, v) in the map, there is a table entry containing (k, v).
- SlotsConsistent says that for each entry with a key-value pair (k, v), that pair is in the map.
- Contiguous says that each key in the table, there is a contiguous range of nonempty entries from the key's hash to that entry.

Now set,

$$m(\gamma, k, v) \triangleq ([k \mapsto \text{ex}(v)], []) \qquad slot(\gamma, i, s) \triangleq ([], [i \mapsto \text{ex}(s)])$$

Now we prove the following:

Linear-Probing Hash Table Logic

Propositions: $m(\gamma, k, v)$ $slot(\gamma, i, s)$ (where $\gamma : Name,\ k : Key,\ v : Value^?,\ i : \mathbb{N},\ s : (Key \times Value)^?$)

$$m(\gamma, k, v) * slot(\gamma, j, \text{Some}((k, v_j))) \vdash v = \text{Some}(v_j) \qquad \text{(QueryFound)}$$

$$\frac{k \neq k_{H(k)}, \ldots, k_{i-1}}{m(\gamma, k, v) * slot(\gamma, i, \text{None}) * (\ast_{H(k) \leq j < i}\ slot(\gamma, j, \text{Some}(k_j, v_j))) \vdash v = \text{None}} \text{(QueryNotFound)}$$

$$m(\gamma, k, v) * slot(\gamma, j, \text{Some}((k, v_j))) \Rrightarrow m(\gamma, k, v') * slot(\gamma, j, \text{Some}((k, v'))) \qquad \text{(UpdateExisting)}$$

$$\frac{k \neq k_{H(k)}, \ldots, k_{j-1}}{\begin{array}{l} m(\gamma, k, v) * slot(\gamma, i, \text{None}) * (\ast_{H(k) \leq j < i}\ slot(\gamma, j, \text{Some}(k_j, v_j))) \\ \Rrightarrow m(\gamma, k, v') * slot(\gamma, j, \text{Some}((k, v'))) * (\ast_{H(k) \leq j < i}\ slot(\gamma, j, \text{Some}(k_j, v_j))) \end{array}} \text{(UpdateInsert)}$$

$$m(\gamma, k, v) \wedge slot(\gamma, j, s) \ \vdash\ m(\gamma, k, v) * slot(\gamma, j, s)$$

$$slot(\gamma, b+1, v_{b+1}) \wedge (\ast_{a \leq j \leq b}\ slot(\gamma, j, s_j)) \ \vdash\ (\ast_{a \leq j \leq b+1}\ slot(\gamma, j, s_j)))$$

$$m(\gamma, k, v) \wedge (\ast_{a \leq j \leq b}\ slot(\gamma, j, s_j)) \ \vdash\ m(\gamma, k, v) * (\ast_{a \leq j \leq b}\ slot(\gamma, j, s_j))$$

Specifically,

- We can prove QueryFound and QueryNotFound by PCM-Valid.
- We can prove UpdateExisting and UpdateInsert by PCM-Update.
- We can prove the last three rules by PCM-And.

6 LANGUAGE EXTENSION FOR NON-ATOMIC MEMORY

In this section, we sketch how Leaf can be applied to a language that includes non-atomic memory.

First, we establish the language and its semantics. We focus here only on heap-related aspects of the language. Each read or write operation is annotated with a label, na (non-atomic) or sc (sequentially-consistent atomic). As the names suggest, the atomic read ($!_{\text{sc}}$) and atomic write ($\leftarrow_{\text{sc}}$) are each executed in a single atomic operation. Meanwhile, the non-atomic operations ($!_{\text{na}}$ and $\leftarrow_{\text{na}}$) each take two steps.

$$\begin{aligned} e \;:=\; & \mid e_1 \leftarrow_{\text{sc}} e_2 \mid !_{\text{sc}} e \\ & \mid e_1 \leftarrow_{\text{na}} e_2 \mid !_{\text{na}} e \\ & \mid e_1 \leftarrow_{\text{na'}} e_2 \mid !_{\text{na'}} e \\ & \ldots \end{aligned}$$

The na' operations are "artificial" language elements, used as intermediate states for the operational semantics.

The heap state is given by $\sigma : \mathit{Loc} \xrightarrow{\text{fin}} (\mathit{Value} \times \text{ReadWrite})$, where the set ReadWrite is given by the elements

$$\text{reading}(n) \mid \text{writing}$$

where $n : \mathbb{N}$. This state is used to track the in-progress non-atomic operations. The state writing means that a (single) write is in-progress, while reading(n) means that n reads are in-progress. Of course, this means that reading(0) means that no reads or writes are in-progress.

By tracking this state, our heap semantics are able to reason about *data races*. These occur when either **(i)** a non-atomic read overlaps with a (atomic or non-atomic) write or **(ii)** a non-atomic write overlaps with any other operation. Any such data-race will result in a "stuck state."

The execution semantics are given as follows:

$$(!_{\text{sc}}\ell, \sigma[\ell \mapsto (v, \text{reading}(n))]) \to (v, \sigma[\ell \mapsto (v, \text{reading}(n))])$$

$$(\ell \leftarrow_{\text{sc}} v', \sigma[\ell \mapsto (v, \text{reading}(0))]) \to ((), \sigma[\ell \mapsto (v', \text{reading}(0))])$$

$$(!_{\text{na}}\ell, \sigma[\ell \mapsto (v, \text{reading}(n))]) \to (!_{\text{na'}}\ell, \sigma[\ell \mapsto (v, \text{reading}(n+1))])$$
$$(!_{\text{na'}}\ell, \sigma[\ell \mapsto (v, \text{reading}(n+1))]) \to (v, \sigma[\ell \mapsto (v, \text{reading}(n))])$$

$$(\ell \leftarrow_{\text{na}} v', \sigma[\ell \mapsto (v, \text{reading}(0))]) \to (\ell \leftarrow_{\text{na'}} v', \sigma[\ell \mapsto (v, \text{writing})])$$
$$(\ell \leftarrow_{\text{na'}} v', \sigma[\ell \mapsto (v, \text{writing})]) \to ((), \sigma[\ell \mapsto (v', \text{reading}(0))])$$

If we cannot execute the appropriate rule because the ReadWrite state is wrong, we result in a stuck state.

We want to prove the usual Hoare triples for both sc and na operations.

HEAP-WRITE-SC
$\{\ell \hookrightarrow v\}\ !_{\text{sc}}\ell\ \{r.\ \ell \hookrightarrow v * v = r\}_{\mathcal{E}_{\text{heap}}}$

HEAP-READ-SHARED-SC
$[\ell \hookrightarrow v]_{\mathcal{E}_{\text{heap}}}\ \{\}\ !_{\text{sc}}\ell\ \{v'.\ v = v'\}_{\mathcal{E}_{\text{heap}}}$

HEAP-WRITE-NA
$\{\ell \hookrightarrow v\}\ !_{\text{na}}\ell\ \{r.\ \ell \hookrightarrow v * v = r\}_{\mathcal{E}_{\text{heap}}}$

HEAP-READ-SHARED-NA
$[\ell \hookrightarrow v]_{\mathcal{E}_{\text{heap}}}\ \{\}\ !_{\text{na}}\ell\ \{v'.\ v = v'\}_{\mathcal{E}_{\text{heap}}}$

From the perspective of the program logic, the only difference between sc and na operations is that the na operations not physically atomic.

Regardless, we are here to prove the Hoare triples. Keep in mind that proving that the triples requires us to prove that the operations do not get stuck when we have the appropriate resources. The key challenge here is that the "read" operations are actually effectful, temporarily updating the ReadWrite at the given location. Therefore, it seems difficult to make use of a shared $\ell \hookrightarrow v$. The trick is to define $\mathcal{H}(\sigma)$ and $\ell \hookrightarrow v$ the right way.

First, define the set ReadWrite' with the elements

$$\mathsf{reading} \mid \mathsf{writing}$$

Define $t : \mathsf{ReadWrite} \to \mathsf{ReadWrite'}$ by,

$$t(\mathsf{reading}(n)) \triangleq \mathsf{reading}$$
$$t(\mathsf{writing}) \triangleq \mathsf{writing}$$

Essentially, this tracks the writing versus non-writing states, while "forgetting" the heap reference count. Now define $T : (\mathit{Loc} \xrightarrow{\text{fin}} (\mathit{Value} \times \mathsf{ReadWrite})) \to (\mathit{Loc} \xrightarrow{\text{fin}} (\mathit{Value} \times \mathsf{ReadWrite'}))$ by mapping t over each ReadWrite.

Now we use the authoritative-fragmentary construction,

$$\textsc{Auth}(\mathit{Loc} \xrightarrow{\text{fin}} (\mathit{Value} \times \mathsf{ReadWrite'})) \text{ for ghost state at } \gamma_{\text{heap}},$$

and the (higher order) construction,

$$\textsc{Auth}((\mathit{Loc} \times \mathbb{N}) \xrightarrow{\text{fin}} \blacktriangleright(\mathit{iProp})) \text{ for ghost state at } \gamma_{\text{reads}}.$$

Now define,

$$\ell \hookrightarrow v \triangleq \boxed{\circ\, [\ell \mapsto (v, \mathsf{reading})]}^{\gamma_{\text{heap}}}$$
$$\ell \overset{\mathsf{w}}{\hookrightarrow} v \triangleq \boxed{\circ\, [\ell \mapsto (v, \mathsf{writing})]}^{\gamma_{\text{heap}}}$$
$$\mathcal{H}(\sigma) \triangleq \exists (m : (\mathit{Loc} \times \mathbb{N}) \xrightarrow{\text{fin}} \mathit{iProp}).$$
$$\boxed{\bullet\, T(\sigma)}^{\gamma_{\text{heap}}} * \boxed{\bullet\, \mathsf{next}(m)}^{\gamma_{\text{reads}}} * \mathit{countsAgree}(\sigma, m) * \mathop{\Large\ast}_{((\ell,i),G) \in m} G * \exists v.\ (G \twoheadrightarrow \mathcal{E}_{\text{heap}}\ (\ell \hookrightarrow v))$$

Here, $\mathit{countsAgree}(\sigma, m)$ means that for each ℓ, the number of entries of the form $(\ell, \_)$ in m is equal to the read-count for ℓ in σ.

The point of all this is that, when we are in the middle of a non-atomic read with a guard proposition G, we can save G in $\mathcal{H}(\sigma)$ during the intermediate step.

Now, observe that:

- When we have a shared $\ell \hookrightarrow v$, we can deduce that $\sigma(\ell)$ is not in the writing state.
- When we have exclusive access to $\ell \hookrightarrow v$, we can further deduce that $\sigma(\ell)$ is in the reading(0) state. To do this, assume by contradiction we are in the reading(n) state for $n > 0$. Then, there is at least one ℓ-entry in m, so we can obtain a guard proposition G where $G \twoheadrightarrow (\ell \hookrightarrow v)$. This is a contradiction because we have exclusive ownership of $\ell \hookrightarrow v$.

This allows us to prove that we do not reach a stuck state when performing the operations.

For the proofs:

- To prove Heap-Write-NA, we perform the first execution step, moving from reading(0) into writing, and exchanging $\ell \hookrightarrow v$ for $\ell \overset{\mathsf{w}}{\hookrightarrow} v$. In the second execution step, we switch back and re-obtain $\ell \hookrightarrow v$.
- To prove Heap-Read-Shared-NA, recall first that it is equivalent to,

  $$\{G * (G \twoheadrightarrow \mathcal{E}_{\text{heap}}\ (\ell \hookrightarrow v))\}\ !_{\mathsf{na}}\ell\ \{v'.\ (v = v') * G\}_{\mathcal{E}_{\text{heap}}}$$

  To show this, we first pick some arbitrary fresh index i and add $[(\ell, i) \mapsto G]$ to the map m. Since we are moving from reading(n) to reading($n + 1$), this means the counts still agree.

Furthermore, it means we exchange G for $\boxed{\circ[(\ell, i) \mapsto \mathsf{next}(G)]}^{\gamma_{\mathsf{reads}}}$. This lets us "remember" what G is, so we can re-obtain G when we perform the second step.

Of course, Heap-Write-SC and Heap-Read-Shared-SC are simpler to prove, since the corresponding heap operations are atomic.