Here, we describe a PCM-based program logic for reasoning about concurrent programs, with a built-in notion of "PCM refinement" to provide modularity that is both useful in its own right and particularly useful for reasoning about "read-only" state.

Monoid definitions

We start by defining what it means to be a PCM with a transition relation $\mapsto$. We call this a *Transitional PCM* (TPCM).

Let $\widehat{M} = (M, \cdot, \mapsto)$ where where $\cdot : M \times M \to M^?$ and $\mapsto : M \times M \to \mathsf{bool}$. (We use $M^? \triangleq M \cup \{\bot\}$.)

We say that $\mapsto$ is *monotonic* if

$$\forall a, b, c.\ (a \mapsto b) \land (a \cdot c \neq \bot) \implies (b \cdot c \neq \bot) \land (a \cdot c \mapsto b \cdot c)$$

This allows us to define what it means to be a TPCM,

$$\frac{\begin{array}{c}\widehat{M} = (M, \cdot, \mapsto) \\ \cdot : M \times M \to M^?,\ \text{comm., assoc.} \\ \mapsto : M \times M \to \mathsf{bool},\ \text{trans., refl., monotonic}\end{array}}{\text{TPCM}(\widehat{M})}\ (\text{TPCM})$$

Note that we can always construct a TPCM by picking $\cdot$ to be commutative and associative, and then by definition setting $\mapsto$ to be,

$$a \mapsto b = \forall c.\ a \cdot c \neq \bot \implies b \cdot c \neq \bot$$

However, we can also set $\mapsto$ to be more restrictive, which will be useful later.

TPCM Example

Consider a *Sharded State Machine* (SSM). (This is similar to the STS common in the literature but framed a little differently.)

An SSM is given by **(i)** some PCM state $S, \cdot$, **(ii)** an invariant $inv : S \to \mathsf{bool}$ and a sharded transition $\rightsquigarrow : S \times S \to \mathsf{bool}$. For a sharded state machine, the invariant inductivity predicate is given by,

$$inv(a \cdot c) \land (a \rightsquigarrow b) \implies inv(b \cdot c)$$

(Note: $\rightsquigarrow$ is neither transitive nor required to be monotonic here. However, an alternative definition would require $\rightsquigarrow$ to be monotonic, in which case we could simplify the inductivity predicate to $inv(a) \land (a \rightsquigarrow b) \implies inv(b)$.)

We can define a TPCM to correspond to this SSM by,

$$M \triangleq \{m \in S : \exists c.\ inv(m \cdot c)\}$$

$$a \cdot_M b \triangleq \begin{cases} a \cdot b & \text{if } a \cdot b \in M \\ \bot, & \text{otherwise} \end{cases}$$

and finally we can define $\mapsto$ as the transitive closure of $\mapsto_{\text{pre}}$ defined as,

$$a \mapsto_{\text{pre}} b \triangleq \exists a_0, b_0, c.\ (a = a_0 \cdot c) \land (b = b_0 \cdot c) \land a_0 \rightsquigarrow b_0$$

TPCM refinements (motivation)

There are a handful of reasons to want to modularize PCM definitions:

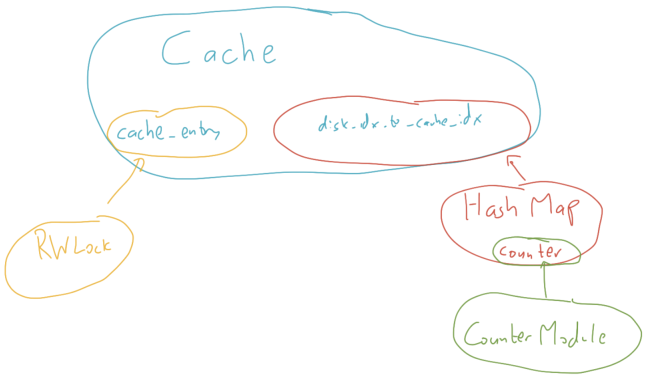
- The "base language" may provide some "points to" assertion $\ell \hookrightarrow v$, and then the user wants to build something else on top of that (like a fractional permission logic or similar) without needing to inject it into the native definition.
- You have some PCM N (custom built for your application), and you want to build a reader-writer lock to manage access to the state of N, without encoding the reader-writer lock protocol into N.
- You have some PCM N, and you want to manage part of it via some hash table implementation. You want to verify the hash table implementation independently (e.g., via some SSM strategy) and then use it modularly to manage some part of the N-state.

- You have some PCM N with a counter field, and you want to write the counter-manipulation code as its own independent module.

These are all things that have come up in VeriBetrKV.

We'll develop the concept of a TPCM refinement, denoted $\widehat{M} \twoheadrightarrow \widehat{N}$. In a modular design, $\widehat{M}$ will be the "sub-component", while $\widehat{N}$ is the client.

For example, a hypothetical cache architecture might have a "main" ghost state Cache represented as an SSM. We might have a sub-component RwLock $\twoheadrightarrow$ Cache, another sub-component HashTable $\twoheadrightarrow$ Cache, and even further a sub-component Counter $\twoheadrightarrow$ HashTable.



TPCM refinements

We'll use the symbol $\twoheadrightarrow$ to denote refinement. Yeah, you can tell I scoured the LaTeX documentation for this thing, but I actually like it: it suggests that the thing on the left is more "broken up" (e.g., like in a fractional logic).

We use $\widehat{M} \twoheadrightarrow_{\text{rel}} \widehat{N}$ to denote that $\widehat{M}$ *refines* $\widehat{N}$ via a relation rel $: M \times N \to bool$. Of course, it must satisfy some conditions (and this is why we allow for flexibility in the definition of $\mapsto$ above).

$$\frac{\begin{array}{c}\text{TPCM}(\widehat{M}) \quad \text{TPCM}(\widehat{N}) \\ \text{rel} : M \times N \to \text{bool} \\ \text{rel}(\epsilon, \epsilon) \\ (\forall b, b', q.\ b \mapsto b' \wedge \text{rel}(b, q) \implies \exists q'.\ \text{rel}(b', q') \wedge (q \mapsto q')) \\ \forall b, a, a'.\ \text{rel}(b, a) \wedge \text{rel}(b, a') \implies a \mapsto a' \end{array}}{\widehat{M} \twoheadrightarrow_{\text{rel}} \widehat{N}} \quad (\text{TPCM-Refinement})$$

Using ghost state in programs

We instantiate instances of monoidal ghost state at *locations*. The notation $\boxed{a}^{\gamma}$ means that state a is at location γ. Typically, locations are considered arbitrary values; here, we have to do a little more to handle the relationships between different ghost states.

For example, suppose we have a refinement $\widehat{M} \twoheadrightarrow_{\text{rel}} \widehat{N}$. Perhaps $\widehat{M}$ represents a reader-writer protocol or something similar. Furthermore, let's say we're tracking ghost state $n \in N$ at location γ: we will use the notation $\boxed{n}^{\gamma}$.

Now, we want to use the $\widehat{M}$ protocol to help manage that state. We'll need some rules for "exchanging" the $\widehat{M}$-state with $\widehat{N}$-state. Furthermore, when we manage the $\widehat{M}$-state, we'll need to somehow "remember" that it corresponds to the $\widehat{N}$-state at γ, via the refinement rel. We will track this information in the location value.

Thus, we'll define the location values recursively, reusing the $\twoheadrightarrow$ notation:

$\gamma \ := \ \alpha \mid \alpha \twoheadrightarrow_{\text{rel}} \gamma.$

Here, α denotes any "base location." We will define rules that let us convert $\boxed{n}^{\gamma}$ state to and from $\boxed{m}^{\alpha \twoheadrightarrow_{\text{rel}} \gamma}$ state.

Here are our rules for manipulating TPCM ghost state, which are mostly standard except the ones involving $\twoheadrightarrow$.

$$\mathsf{True} \implies \boxed{\epsilon}^{\gamma}$$

$$\boxed{\bot}^{\gamma} \implies \mathsf{False}$$

$$\boxed{t \cdot u}^{\gamma} \iff \boxed{t}^{\gamma} * \boxed{u}^{\gamma}$$

$$\frac{\text{TPCM}(\widehat{M}) \quad a \in M}{\mathsf{True} \Rrightarrow \exists \alpha.\, \boxed{a}^{\alpha}} \; (\text{GHOST-ALLOC})$$

$$\frac{\begin{array}{c}\widehat{M} \twoheadrightarrow_{\text{rel}} \widehat{N} \\ \text{rel}(m, n)\end{array}}{\boxed{n}^{\gamma} \Rrightarrow \exists \alpha.\, \boxed{m}^{\alpha \twoheadrightarrow_{\text{rel}} \gamma}} \; (\text{GHOST-REFINEMENT-ALLOC})$$

$$\frac{a \mapsto b}{\boxed{a}^{\gamma} \Rrightarrow \boxed{b}^{\gamma}} \; (\text{GHOST-VIEWSHIFT})$$

$$\frac{\begin{array}{c}\widehat{M} \twoheadrightarrow_{\text{rel}} \widehat{N} \\ \forall p, q.\ \text{rel}(b \cdot p) = \mathit{Some}(q) \implies \exists q'.\ \text{rel}(b' \cdot p) = \mathit{Some}(q') \land (a \cdot q \mapsto a' \cdot q')\end{array}}{\boxed{b}^{\alpha \twoheadrightarrow_{\text{rel}} \gamma} * \boxed{a}^{\gamma} \Rrightarrow \boxed{b'}^{\alpha \twoheadrightarrow_{\text{rel}} \gamma} * \boxed{a'}^{\gamma}} \; (\text{GHOST-REFINEMENT-EXCHANGE})$$

Justification.

To justify these rules, we can roll up all the $\boxed{m}^{\gamma}$ values into a mega-monoid (or resource algebra) one defined as a mapping from locations to monoid elements. Then we show that all those rules above follow from frame-preserving updates.

Notes about abstraction.

When using a refinement $\widehat{M} \twoheadrightarrow \widehat{N}$ to abstract out some subcomponent, we observe that $\widehat{M}$ (which corresponds to the subcomponent) needs to be defined with respect to the base state $\widehat{N}$, which initially seems a little strange. Shouldn't we define the subcomponent *first*, and then build the larger components out of those?

And yes, if for example $\widehat{M}$ is a reader-writer lock, then $\widehat{M}$ and the refinement rel will need to be defined in terms of $\widehat{N}$. However, this is not really so different than using a templated type, e.g., writing `RwLock<T>` to create a mutex storing type `T`.

Borrows and read-only state.

I think we can use a fairly simple system for representing (shared) borrows. We use κ to represent a lifetime variable. $\&^{\kappa} A$ represents some borrowed proposition; active(κ) indicates that lifetime κ is active, and borrowed(κ, A) indicates you can get back exclusive access to A once the lifetime κ expires.

We use $\kappa_1 \sqcap \kappa_2$ to represent lifetime intersection and $\kappa_1 \sqsubseteq \kappa_2$ to represent lifetime inclusion.

Of course, the intent is that the verification framework will use some sort of borrow-checker to identify the lifetimes and perform these automatically.

(Note: RustBelt uses a much more complex system - need to understand why; am I missing something or is it for some feature I don't need?)

$$\frac{}{A \Rrightarrow \exists \kappa.\ \text{active}(\kappa) * \text{borrowed}(\kappa, A) * \&^{\kappa} A} \; (\text{BORROW-BEGIN})$$

$$\frac{}{\text{active}(\kappa) * \text{borrowed}(\kappa, A) \Rrightarrow A} \ (\textsc{Borrow-end})$$

$$\frac{}{\&^{\kappa} A \Rrightarrow \&^{\kappa} A * \&^{\kappa} A} \ (\textsc{Borrow-dupe})$$

$$\frac{\kappa_1 \sqsupseteq \kappa_2}{\&^{\kappa_1} A \Rrightarrow \&^{\kappa_2} A} \ (\textsc{Reborrow})$$

$$\text{active}(\kappa_1 \sqcap \kappa_2) \iff \text{active}(\kappa_1) * \text{active}(\kappa_2)$$

Next, let's add some rules for making use of borrowed TPCM state.

$$\mathsf{True} \implies \&^{\kappa}\boxed{\epsilon}^{\gamma}$$

$$\text{active}(\kappa) * \&^{\kappa}\boxed{\bot}^{\gamma} \implies \mathsf{False}$$

$$\frac{\forall r.\ m \le r \wedge n \le r \implies k \le r}{\&^{\kappa}\boxed{m}^{\gamma} * \&^{\kappa}\boxed{n}^{\gamma} \Rrightarrow \&^{\kappa}\boxed{k}^{\gamma}} \ (\textsc{Borrow-Combine})$$

$$\frac{c \cdot a \mapsto c \cdot b}{\text{active}(\kappa) * \&^{\kappa}\boxed{c}^{\gamma} * \boxed{a}^{\gamma} \Rrightarrow \text{active}(\kappa) * \boxed{b}^{\gamma}} \ (\textsc{Borrow-Ghost-ViewShift})$$

Finally, rules to allow borrows to interact with TPCM refinements:

$$\frac{\widehat{M} \twoheadrightarrow_{\text{rel}} \widehat{N} \qquad \forall p, b.\ \text{rel}(m \cdot p, b) \implies a \le b}{\&^{\kappa}\boxed{m}^{\delta \twoheadrightarrow \gamma} \Rrightarrow \&^{\kappa}\boxed{a}^{\gamma}} \ (\textsc{Borrow-Back})$$

$$\frac{\widehat{M} \twoheadrightarrow_{\text{rel}} \widehat{N} \qquad \forall p, q.\ \text{rel}(c \cdot b \cdot p, q) \implies \exists q'.\ \text{rel}(c \cdot b' \cdot p, q') \wedge (a \cdot q \mapsto a' \cdot q')}{\text{active}(\kappa) * \&^{\kappa}\boxed{c}^{\delta \twoheadrightarrow \gamma} * \boxed{b}^{\delta \twoheadrightarrow \gamma} * \boxed{a}^{\gamma} \Rrightarrow \text{active}(\kappa) * \boxed{b'}^{\delta \twoheadrightarrow \gamma} * \boxed{a'}^{\gamma}} \ (\textsc{Borrow-Ghost-Refinement-Exchange})$$

Note that Borrow-Ghost-Refinement-Exchange allows us to utilize a borrow in the extension world $\delta \twoheadrightarrow \gamma$ but *not* a borrow in the base world γ. This is important as we would have an unsoundness otherwise. In general, we can't have any rule that allows a borrow in the γ world to flow back to the $\delta \twoheadrightarrow \gamma$ world, as then that borrow could "interfere" with the $\delta \twoheadrightarrow \gamma$-world's state which is being held "in reserve" for that borrow.

Justification

As before, our justification will be to roll all the ghost state, the $\&^{\kappa}\boxed{a}^{\gamma}$ tokens, the active tokens, and the borrowed tokens into a single mega-monoid, and derive the above rules as frame-preserving updates.

There's one trick worth pointing out. Intuitively, we're going to have some invariant that says the following:

Suppose we have $\text{active}(\kappa) * \&^{\kappa}\boxed{a}^{\gamma}$ with $a \in M$. Then, consider:

- All "reserved" γ-state: $\text{borrowed}(\kappa', \boxed{b}^{\gamma})$

- All state in an extension δ of γ, both *borrowed and non-borrowed*. By "extension" δ, we mean any location $\alpha \rightarrowtail \gamma$, $\alpha \rightarrowtail (\beta \rightarrowtail \gamma)$, and so on, (and *not* including γ itself). Project all this state down to M via the refinement relations.

Borrow state invariant: Then the monoidal sum of all this state gives some value t where $a \preceq t$.

Essentially, this says that for active borrow state $\&^{\kappa}\boxed{a}^{\gamma}$, that state is actually "backed" by some state which is either "borrowed out" or in some "refinement world."

Note again in the second bullet that within the "refinement worlds" we allow both *borrowed and non-borrowed* state. This is the "trick" that lets the TPCM refinement be so effective for dealing with borrowed state.

For example, suppose we derive the fractional logic $\mathrm{Frac}(\widehat{N}) \rightarrowtail \widehat{N}$.

Now, as a user of the fractional logic, we can derive $a \Rrightarrow \mathrm{Frac}(1, a)$, and then $\mathrm{Frac}(1, a) \Rrightarrow \mathrm{Frac}(1/2, a) * \mathrm{Frac}(1/2, a)$. Then, we could perform a borrow of one of the halves and obtain $\&^{\kappa}\,\mathrm{Frac}(1/2, a)$, then obtain $\&^{\kappa} a$.

We are able to do this, even though we have "only" borrowed half, e.g., we have $\mathrm{borrowed}(\kappa, \mathrm{Frac}(1/2, a)) * \mathrm{Frac}(1/2, a)$, rather than $\mathrm{borrowed}(\kappa, \mathrm{Frac}(1, a))$.

Justification (more formally)

We can define a resource algebra (RA) as follows. Let Γ be the set of locations γ and K the set of lifetimes κ.

- *live* state at each location γ given by $live : \Gamma \to m$
- *reserved* state at each location γ and lifetime κ, given by $reserved : K \to \Gamma \to m$.
- A set of *borrows* where $borrows \subseteq K \times \Gamma \times M$.
- An active set of lifetimes, given by $active : K$.

The live state is additive (that is, $live(\gamma \to m) \cdot live(\gamma \to p) \triangleq live(\gamma \to p \cdot q)$. Reserved state is similarly additive.

Finally, borrows are freely duplicable, and add *freely* (i.e., take the union of the set of borrow objects).

We'll describe the invariants on this resource algebra below, but first, I'll explain how all the above concepts are embedded into this resource algebra:

$$\begin{aligned}
\boxed{m}^{\gamma} &\triangleq live(\gamma \to m) \\
\mathrm{borrowed}(\kappa, \boxed{m}^{\gamma}) &\triangleq reserved(\kappa, \gamma \to m) \\
\&^{\kappa}\boxed{m}^{\gamma} &\triangleq borrow(\kappa, \gamma \to m) \\
\mathrm{active}(\kappa) &\triangleq active(\kappa)
\end{aligned}$$

Now we define some functions on this state, defined recursively. First, for any refinement location $\delta \rightarrowtail_{\mathrm{rel}} \gamma$, where $\mathrm{rel} : M \times N \to \mathrm{bool}$, we let $Project_{\delta} : M \to N$ be a "functionalization" of rel. In particular,

$$\forall m.\ m \in domain(\mathrm{rel}) \implies \mathrm{rel}(m, Project_{\delta}(m)).$$

where $domain(\mathrm{rel}) \triangleq \{m : \exists n.\ \mathrm{rel}(m, n)\}$.

Now we define $Total_{\kappa}(\gamma)$ to be the "total" state of a location γ: that is, all the live and reserved state (for the active livetime κ) *plus* all the total state of locations which depend on γ. Thus we use this recursive definition,

$$Total_{\kappa}(\gamma) \triangleq live(\gamma) \cdot \left(\bigodot_{\kappa' \sqsupseteq \kappa} reserved(\kappa', \gamma)\right) \cdot \left(\bigodot_{\delta} Project_{\delta}\left(Total_{\kappa}(\delta \rightarrowtail_{\mathrm{rel}} \gamma)\right)\right)$$

Now, this definition is only sensible when $Total_{\kappa}(\delta \rightarrowtail_{\mathrm{rel}} \gamma)$ is in the "domain" of rel.

Thus we will have an invariant, only applying to the active lifetime *active*,

$$Total_{active}(\delta \rightarrowtail_{\mathrm{rel}} \gamma) \in domain(\mathrm{rel}) \tag{1}$$

Next, we define a similar recursive function called $View$, which will be defined only in terms of the *reserved* state. However, it will be used to explain what the "possible" values of the $Total$ (minus the live state). A view will be a function $v : M \to \text{bool}$. These views will also be closed *upwards*, i.e., $v(a) \implies v(a \cdot b)$.

Again, we define a "projection" function from a view on a refinement TPCM to a base TPCM.

$$ProjectView_\delta(v) \triangleq \lambda a.\ \exists x, y.\ v(x) \wedge y = Project_\delta(x) \wedge (x \in domain(rel)) \wedge y \leq a.$$

For a given m, we define $ViewOver(m)$ to be the view of all elements that are at least m:

$$ViewOver(m) \triangleq \lambda n.\ n \geq m$$

And we can add two views in the natural way:

$$v \otimes w \triangleq \lambda m.\ \exists x, y.\ v(x) \wedge w(y) \wedge x \cdot y = m$$

Finally, we're ready to define the $View_\kappa$ function for a given location, again recursively, like the $Total_\kappa$ function:

$$View_\kappa(\gamma) \triangleq \left(\bigotimes_{\kappa' \sqsupseteq \kappa} ViewOver(reserved(\kappa', \gamma)) \right) \otimes \left(\bigotimes_{\delta} ProjectView_\delta \left(View_\kappa(\delta \twoheadrightarrow_{\text{rel}} \gamma) \right) \right)$$

Now the $View_\kappa$ denotes the "possible" states that the total unlive state can be in for a given γ, without actually depending on any live state even in the refining locations. The $View_\kappa$ is in some sense used to justify the borrows at κ, so we will have an invariant,

$$\forall \kappa, \gamma, m.\ (\kappa, \gamma, m) \in borrows \implies \forall y.\ View_\kappa(\gamma)(y) \implies m \leq y \tag{2}$$

(We don't need a condition that $\kappa \sqsupseteq active$, although we won't be able to make use of the invariant otherwise.)

Now, we define Inv to be both (1) and (2) together, and define an RA with a validity predicate $\mathcal{V}(x) \triangleq \exists y.\ Inv(x \cdot y)$. Then we can derive all the rules claimed above, by simply showing that they are frame-preserving within this RA.

First, the borrowing rules: we can check that they preserve (2). Also, all of the borrowing rules will preserve $Total_{active}$, so they preserve (1) trivially.

Now, for the frame update rules (Borrow-Ghost-ViewShift and Borrow-Ghost-Refinement-Exchange). First, these don't affect $View_\kappa$ at all, so it is easy to show that they preserve (2).

We can also check that (1) is preserved. There are two key points here. First, the rules about TPCM refinements imply that a frame-preserving update on a $Total_\kappa(\delta \twoheadrightarrow_{\text{rel}} \gamma)$ implies a frame-preserving update on the projection, $Project_\delta(Total_\kappa(\delta \twoheadrightarrow_{\text{rel}} \gamma))$.

Second, we have to show why it is allowed to use borrowed state to justify the transitions. Luckily, from (1) we can derive this law,

$$\kappa \sqsupseteq active \implies View_\kappa(\gamma) \left(\left(\bigcirc_{\kappa' \sqsupseteq active} reserved(\kappa, \gamma) \right) \cdot \left(\bigcirc_\delta Project_\delta \left(Total_{active}(\delta \twoheadrightarrow_{\text{rel}} \gamma) \right) \right) \right).$$

This large expression inside the $View_\kappa(\gamma)$ is just the part of $Total_{active}(\gamma)$ except the part *which is live at* γ. Thus, combined with (2), we can see that any borrow will be covered by the reserved state at γ plus all state in the refined locations.

Just to reiterate the "trick": $View_\kappa(\gamma)$ depends only on reserved state: however we are able to show that the value

$$\left(\bigcirc_{\kappa' \sqsupseteq active} reserved(\kappa', \gamma) \right) \cdot \left(\bigcirc_\delta Project_\delta \left(Total_{active}(\delta \twoheadrightarrow_{\text{rel}} \gamma) \right) \right)$$

is in the active view, even though it includes the *total* state at locations refining γ.

Fractional logic

Given a TPCM M, we can construct a TPCM $Frac_M$ with the following laws.

$$\boxed{x}^{\gamma} \Lleftarrow\!\Rrightarrow \boxed{\mathsf{Frac}(x, 1)}^{\alpha \rightarrowtail \gamma}$$

$$\boxed{\mathsf{Frac}(x, r)}^{\alpha \rightarrowtail \gamma} * \boxed{\mathsf{Frac}(x, q)}^{\alpha \rightarrowtail \gamma} \Lleftarrow\!\Rrightarrow \boxed{\mathsf{Frac}(x, r + q)}^{\alpha \rightarrowtail \gamma}$$

And,

$$\&^{\kappa} \boxed{\mathsf{Frac}(x, r)}^{\alpha \rightarrowtail \gamma} \Rrightarrow \&^{\kappa} \boxed{x}^{\gamma}$$

Counting logic

$$\boxed{x}^{\gamma} \Lleftarrow\!\Rrightarrow \boxed{\mathsf{Counter}(x, 0)}^{\alpha \rightarrowtail \gamma}$$

$$\boxed{\mathsf{Counter}(x, n)}^{\alpha \rightarrowtail \gamma} \Lleftarrow\!\Rrightarrow \boxed{\mathsf{Counter}(x, n + 1)}^{\alpha \rightarrowtail \gamma} * \boxed{\mathsf{SharedRef}(x)}^{\alpha \rightarrowtail \gamma}$$

And,

$$\&^{\kappa} \boxed{\mathsf{SharedRef}(x)}^{\alpha \rightarrowtail \gamma} \Rrightarrow \&^{\kappa} \boxed{x}^{\gamma}$$

RwLock Logic

Given a TPCM M, we can construct a TPCM $RwLock_M$ with the following laws. (Here, we have exc : bool, $rc \in \mathbb{N}$, and $x \in M$.

These laws are for exclusive locks:

$$\boxed{\mathsf{Central}\,(\mathsf{False}, rc, x)}^{\alpha \rightarrowtail \gamma} \Rrightarrow \boxed{\mathsf{Central}\,(\mathsf{True}, rc, x)}^{\alpha \rightarrowtail \gamma} * \boxed{\mathsf{ExcPending}}^{\alpha \rightarrowtail \gamma} \qquad \text{(exc-begin)}$$

$$\boxed{\mathsf{Central}\,(exc, 0, x)}^{\alpha \rightarrowtail \gamma} * \boxed{\mathsf{ExcPending}}^{\alpha \rightarrowtail \gamma} \Rrightarrow \boxed{\mathsf{Central}\,(exc, 0, x)}^{\alpha \rightarrowtail \gamma} * \boxed{\mathsf{ExcGuard}}^{\alpha \rightarrowtail \gamma} * \boxed{x}^{\gamma} \qquad \text{(exc-finish)}$$

$$\boxed{\mathsf{Central}\,(exc, rc, y)}^{\alpha \rightarrowtail \gamma} * \boxed{\mathsf{ExcGuard}}^{\alpha \rightarrowtail \gamma} * \boxed{x}^{\gamma} \Rrightarrow \boxed{\mathsf{Central}\,(\mathsf{False}, rc, x)}^{\alpha \rightarrowtail \gamma} \qquad \text{(exc-return)}$$

The first two are for acquisition, while the third is for release. Meanwhile, the first one is derived from Ghost-ViewShift, while the last two are derived from Ghost-Refinement-Exchange.

These laws are for shared locks:

$$\boxed{\mathsf{Central}\,(exc, rc, x)}^{\alpha \rightarrowtail \gamma} \Rrightarrow \boxed{\mathsf{Central}\,(exc, rc + 1, x)}^{\alpha \rightarrowtail \gamma} * \boxed{\mathsf{SharedPending}}^{\alpha \rightarrowtail \gamma} \qquad \text{(shared-begin)}$$

$$\boxed{\mathsf{Central}\,(exc, rc, x)}^{\alpha \rightarrowtail \gamma} * \boxed{\mathsf{SharedPending}}^{\alpha \rightarrowtail \gamma} \Rrightarrow \boxed{\mathsf{Central}\,(exc, rc - 1, x)}^{\alpha \rightarrowtail \gamma} \qquad \text{(shared-abort)}$$

$$\boxed{\mathsf{Central}\,(false, rc, x)}^{\alpha \rightarrowtail \gamma} * \boxed{\mathsf{SharedPending}}^{\alpha \rightarrowtail \gamma} \Rrightarrow \boxed{\mathsf{Central}\,(false, rc, x)}^{\alpha \rightarrowtail \gamma} * \boxed{\mathsf{SharedGuard}(x)}^{\alpha \rightarrowtail \gamma} \qquad \text{(shared-return)}$$

$$\boxed{\mathsf{Central}\,(exc, rc, x)}^{\alpha \rightarrowtail \gamma} * \boxed{\mathsf{SharedGuard}(x)}^{\alpha \rightarrowtail \gamma} \Rrightarrow \boxed{\mathsf{Central}\,(exc, rc - 1, x)}^{\alpha \rightarrowtail \gamma} \qquad \text{(shared-abort)}$$

All of these are derived from Ghost-ViewShift.

We don't use Ghost-Refinement-Exchange, because in the 'shared' case, our mechanism for getting back M-state is to use the Borrow-Back rule. In particular, we use it to derive this:

$$\&^{\kappa} \boxed{\mathsf{SharedGuard}(x)}^{\alpha \rightarrowtail \gamma} \Rrightarrow \&^{\kappa} \boxed{x}^{\gamma}$$

This logic is all we need to verify a reader-writer lock.

RwLock Verified Example

Let's use a struct $rwlock : \{exc : Pointer, rc : Pointer\}$. Our implementation will be straightforward:

```
$\text{AcquireExc}(rwlock) \triangleq$ loop until CAS$(rwlock.exc, 0, 1)$;
                        loop until $(!rwlock.rc) = 0$

$\text{AcquireShared}(rwlock) \triangleq$ loop until (
    (loop until $!rwlock.exc$);
    AtomicAdd$(rwlock.rc, 1)$;
    if $(!rwlock.exc) = 0$ then
        True
    else
        AtomicAdd$(rwlock.rc, -1)$; False
  )

$\text{ReleaseExc}(rwlock) \triangleq rwlock.exc \leftarrow 0$

$\text{ReleaseShared}(rwlock) \triangleq$ AtomicAdd$(rwlock.rc, -1)$
```

We would like to prove the following specifications, for some predicate $Inv(rwlock, \alpha, \gamma, inv)$. (Here, inv is the user-defined predicate on the ghost state stored by the $rwlock$.)

AcquireExc:

$$\{Inv(rwlock, \alpha, \gamma, inv)\}$$
$$\text{AcquireExc}(rwlock)$$
$$\left\{Inv(rwlock, \alpha, \gamma, inv) * \boxed{\text{ExcGuard}}^{\alpha \rightarrowtail \gamma} * \exists x. \boxed{x}^{\gamma} \wedge inv(x)\right\}$$

ReleaseExc:

$$\left\{Inv(rwlock, \alpha, \gamma, inv) * \boxed{\text{ExcGuard}}^{\alpha \rightarrowtail \gamma} * \boxed{x}^{\gamma} \wedge inv(x)\right\}$$
$$\text{ReleaseExc}(rwlock)$$
$$\{Inv(rwlock, \alpha, \gamma, inv)\}$$

AcquireShared:

$$\{Inv(rwlock, \alpha, \gamma, inv)\}$$
$$\text{AcquireShared}(rwlock)$$
$$\left\{Inv(rwlock, \alpha, \gamma, inv) * \exists x. \boxed{\text{SharedGuard}(x)}^{\alpha \rightarrowtail \gamma} \wedge inv(x)\right\}$$

ReleaseShared:

$$\left\{Inv(rwlock, \alpha, \gamma, inv) * \boxed{\text{SharedGuard}(x)}^{\alpha \rightarrowtail \gamma} \wedge inv(x)\right\}$$
$$\text{ReleaseShared}(rwlock)$$
$$\{Inv(rwlock, \alpha, \gamma, inv)\}$$

To prove these specifications, we will first define Inv:

$$Inv(rwlock, \alpha, \gamma, inv) \triangleq \exists \iota, \boxed{\exists exc, rc, x. \boxed{\text{Central}(exc, rc, x)}^{\alpha \rightarrowtail \gamma} * (rwlock.exc \hookrightarrow exc) * (rwlock.rc \hookrightarrow rc)}^{\iota}$$

Now, we can work out how to verify the above programs. AcquireExc will follow from the below:

$$\{Inv(rwlock, \alpha, \gamma, inv)\}$$
$$\mathbf{CAS}(rwlock.exc, 0, 1)$$
$$\left\{v.\ Inv(rwlock, \alpha, \gamma, inv) * \left(v \implies \boxed{\mathsf{ExcPending}}^{\alpha \twoheadrightarrow \gamma}\right)\right\}$$

$$\left\{Inv(rwlock, \alpha, \gamma, inv) * \boxed{\mathsf{ExcPending}}^{\alpha \twoheadrightarrow \gamma}\right\}$$
$$(!rwlock.rc)$$
$$\left\{v.\ Inv(rwlock, \alpha, \gamma, inv) * \left(v \neq 0 \implies \boxed{\mathsf{ExcPending}}^{\alpha \twoheadrightarrow \gamma}\right) * \left(v = 0 \implies \boxed{\mathsf{ExcGuard}}^{\alpha \twoheadrightarrow \gamma} * \exists x.\ \boxed{x}^{\gamma} \wedge inv(x)\right)\right\}$$

AcquireShared will follow from:

$$\{Inv(rwlock, \alpha, \gamma, inv)\}$$
$$\mathbf{AtomicAdd}(rwlock.rc, 1) \tag{3}$$
$$\{Inv(rwlock, \alpha, \gamma, inv) * \mathsf{SharedPending}\}$$

$$\{Inv(rwlock, \alpha, \gamma, inv) * \mathsf{SharedPending}\}$$
$$(!rwlock.exc)$$
$$\left\{v.\ Inv(rwlock, \alpha, \gamma, inv) * (v \neq 0 \implies \mathsf{SharedPending}) * \left(v = 0 \implies \exists x.\ \boxed{\mathsf{SharedGuard}(x)}^{\alpha \twoheadrightarrow \gamma} \wedge inv(x)\right)\right\}$$

$$\{Inv(rwlock, \alpha, \gamma, inv) * \mathsf{SharedPending}\}$$
$$\mathbf{AtomicAdd}(rwlock.rc, -1)$$
$$\{Inv(rwlock, \alpha, \gamma, inv)\}$$

ReleaseExc will follow from:

$$\left\{Inv(rwlock, \alpha, \gamma, inv) * \boxed{\mathsf{ExcGuard}}^{\alpha \twoheadrightarrow \gamma} * \boxed{x}^{\gamma} \wedge inv(x)\right\}$$
$$rwlock.exc \leftarrow 0$$
$$\{Inv(rwlock, \alpha, \gamma, inv)\}$$

And finally, ReleaseShared will follow from:

$$\left\{Inv(rwlock, \alpha, \gamma, inv) * \boxed{\mathsf{SharedGuard}(x)}^{\alpha \twoheadrightarrow \gamma} \wedge inv(x)\right\}$$
$$\mathbf{AtomicAdd}(rwlock.rc, -1)$$
$$\{Inv(rwlock, \alpha, \gamma, inv)\}$$

Let's work out (3) explicitly:

$$\{Inv(rwlock, \alpha, \gamma, inv)\}$$
$$\text{(open Inv)} \quad \left\{\exists exc, rc, x.\ \boxed{\mathsf{Central}(exc, rc, x)}^{\alpha \twoheadrightarrow \gamma} * (rwlock.exc \hookrightarrow exc) * (rwlock.rc \hookrightarrow rc)\right\}$$
$$\mathbf{AtomicAdd}(rwlock.rc, 1)$$
$$\left\{\exists exc, rc, x.\ \boxed{\mathsf{Central}(exc, rc, x)}^{\alpha \twoheadrightarrow \gamma} * (rwlock.exc \hookrightarrow exc) * (rwlock.rc \hookrightarrow rc + 1)\right\}$$
$$\text{(shared-begin)} \quad \left\{\exists exc, rc, x.\ \boxed{\mathsf{Central}(exc, rc + 1, x)}^{\alpha \twoheadrightarrow \gamma} * \mathsf{SharedPending} * (rwlock.exc \hookrightarrow exc) * (rwlock.rc \hookrightarrow rc + 1)\right\}$$
$$(rc' = rc + 1) \quad \left\{\exists exc, rc', x.\ \boxed{\mathsf{Central}(exc, rc', x)}^{\alpha \twoheadrightarrow \gamma} * \mathsf{SharedPending} * (rwlock.exc \hookrightarrow exc) * (rwlock.rc \hookrightarrow rc')\right\}$$
$$\text{(close Inv)} \quad \{Inv(rwlock, \alpha, \gamma, inv) * \mathsf{SharedPending}\}$$

A hash-table logic

We'll want to support the following specs for a linear-probing hash table.

$$\left\{ HTInv(ht, \gamma) * \boxed{k \overset{\text{hash}}{\hookrightarrow} v_0}^{\gamma} \right\} \text{Query}(ht, k) \left\{ v.\ HTInv(ht, \gamma) * \boxed{k \overset{\text{hash}}{\hookrightarrow} v_0}^{\gamma} \land v = v_0 \right\}$$

$$\left\{ HTInv(ht, \gamma) * \boxed{k \overset{\text{hash}}{\hookrightarrow} v_0}^{\gamma} \right\} \text{Update}(ht, k, v) \left\{ HTInv(ht, \gamma) * \boxed{k \overset{\text{hash}}{\hookrightarrow} \text{Some } v}^{\gamma} \right\}$$

$$\left\{ HTInv(ht, \gamma) * \boxed{k \overset{\text{hash}}{\hookrightarrow} v_0}^{\gamma} \right\} \text{Remove}(ht, k) \left\{ HTInv(ht, \gamma) * \boxed{k \overset{\text{hash}}{\hookrightarrow} \text{None}}^{\gamma} \right\}$$

We assume a function $hash : Keys \to \mathbb{N}$. We'll let $ht : \{rwlocks : rwlock\ list, table : ((Key, Value)\ option)\ list\}$.

```
$\text{Query}(ht, k) \triangleq \text{QueryIter}(ht, k, hash(k))$
$\text{QueryIter}(ht, k, i) \triangleq \text{AcquireShared}(ht.rwlocks[i]);$
    **let** $r$ = (**match** $!ht.table[i]$ **with**
      | None $\Rightarrow$ None
      | Some $(k_0, v_0) \Rightarrow$ **if** $k = k_0$ **then** $Some(v_0)$ **else** $\text{QueryIter}(ht, k, i + 1)$
    **end**) **in**
    $\text{ReleaseShared}(ht.rwlocks[i]);$
    $r$

$\text{Update}(ht, k, v) \triangleq \text{UpdateIter}(ht, k, v, hash(k))$
$\text{UpdateIter}(ht, k, v, i) \triangleq \text{AcquireExc}(ht.rwlocks[i]);$
    **let** $r$ = (**match** $!ht.table[i]$ **with**
      | None $\Rightarrow ht.table[i] \leftarrow (k, v)$
      | Some $(k_0, v_0) \Rightarrow$ **if** $k = k_0$ **then** $ht.table[i] \leftarrow (k, v)$ **else** $\text{UpdateIter}(ht, k, v, i + 1)$
    **end**) **in**
    $\text{ReleaseExc}(ht.rwlocks[i]);$
    $r$
```

We'll define a TPCM resource for our hash table (that is, a "normal" TPCM, not a refinement). It will have two kinds of elements, $k \overset{\text{hash}}{\hookrightarrow} v$ (where $v : Value\ option$) and $i \overset{\text{entry}}{\hookrightarrow} c$, where $c : (Key, Value)\ option$.

Allowable operations will be encoded into this logic. For example, we will have all the following:

Update.

$$\begin{aligned} &hash(k) \le u \ \wedge \ \big(\forall i.\ hash(k) \le i < u \implies i \in entries \wedge \exists k_1, v_1.\ entries[i] = \mathrm{Some}(k_1, v_1) \wedge k_1 \ne k\big) \\ &\quad \wedge \ \big(entries[u] = \mathrm{None} \ \vee \ \exists v_1.\ entries[u] = \mathrm{Some}(k, v)\big) \\ &\quad \implies entries \cdot (k \overset{\text{hash}}{\hookrightarrow} v_0) \ \mapsto \ entries[u \leftarrow \mathrm{Some}\ (k, v)] \cdot (k \overset{\text{hash}}{\hookrightarrow} v) \end{aligned}$$

Query (found).

The found case is easy: a single entry (k, v) proves that the hash table maps k to v.

$$\left((i \overset{\text{entry}}{\hookrightarrow} (k, v_0)) \cdot (k \overset{\text{hash}}{\hookrightarrow} v)\right) \ne \bot \implies v = \mathrm{Some}(v_0)$$

The not-found case requires another quantifed predicate:

$$\begin{aligned} &\big(\forall i.\ hash(k) \le i < u \implies i \in entries \wedge \exists k_1, v_1.\ entries[i] = \mathrm{Some}(k_1, v_1) \wedge k_1 \ne k\big) \\ &\quad \wedge \ (entries[u] = \mathrm{None}) \\ &\quad \implies entries \cdot (k \overset{\text{hash}}{\hookrightarrow} v) \ne \bot \implies v = \mathrm{None} \end{aligned}$$

We'll define $HTInv$ as follows:

$$\begin{aligned} HTInv(ht, \gamma) &\triangleq \exists \alpha.\ \mathop{\Large *}_{i} Inv\,(ht.rwlocks[i], \alpha, \gamma \times Heap, htinv(i, ht.entries[i])) \\ htinv(i, \ell) &\triangleq \lambda x.\ \exists c.\ x = (i \overset{\text{entry}}{\hookrightarrow} c, \ell \hookrightarrow c) \end{aligned}$$

In other words, the RwLocks will be arranged so that we can use the lock at index i to obtain some ghost state $(i \overset{\text{entry}}{\hookrightarrow} c, ht.entries[i] \hookrightarrow c)$.

Now, we prove QueryIter. (To avoid clutter, we silently skolemize existential variables as they are introduced.)

$$\left\{ \mathsf{active}(\kappa) * \&^{\kappa}\boxed{ht.entries}^{\gamma} * \boxed{k \overset{\text{hash}}{\hookrightarrow} v_0}^{\gamma} \wedge KeyNotInRange(ht.entries, k, i) \right\}$$

AcquireShared($ht.rwlocks[i]$);

$$\left\{ \mathsf{active}(\kappa) * \&^{\kappa}\boxed{ht.entries}^{\gamma} * \boxed{k \overset{\text{hash}}{\hookrightarrow} v_0}^{\gamma} \wedge KeyNotInRange(ht.entries, k, i) * \mathsf{SharedGuard}\left((i \overset{\text{entry}}{\hookrightarrow} c, ht.entries[i] \hookrightarrow c) \right) \right\}$$

$$\left\{ \mathsf{active}(\kappa) * \&^{\kappa}\boxed{ht.entries}^{\gamma} * \boxed{k \overset{\text{hash}}{\hookrightarrow} v_0}^{\gamma} \wedge KeyNotInRange(ht.entries, k, i) \right.$$
$$\left. * \mathsf{active}(\kappa_0) * \mathsf{borrowed}\left(\kappa_0, \mathsf{SharedGuard}\left((i \overset{\text{entry}}{\hookrightarrow} c, ht.entries[i] \hookrightarrow c) \right) \right) * \&^{\kappa_0}\mathsf{SharedGuard}\left((i \overset{\text{entry}}{\hookrightarrow} c, ht.entries[i] \hookrightarrow c) \right) \right\}$$

```
**let** $r$ = (**match** $!ht.table[i]$ **with**
    | None ⇒ None
    | Some $(k_0, v_0)$ ⇒ **if** $k = k_0$ **then** $Some(v_0)$ **else** QueryIter($ht, k, i + 1$)
**end**) **in**
ReleaseShared($ht.rwlocks[i]$);
$r$
```

$$\left\{ \mathsf{active}(\kappa) * (k \overset{\text{hash}}{\hookrightarrow} v_0) \wedge v = v_0 \right\}$$