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 $\mapsto$.

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 PCM,

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

Note that we can always construct a PCM 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.

PCM 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 PCM 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,

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

PCM refinements (motivation)

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

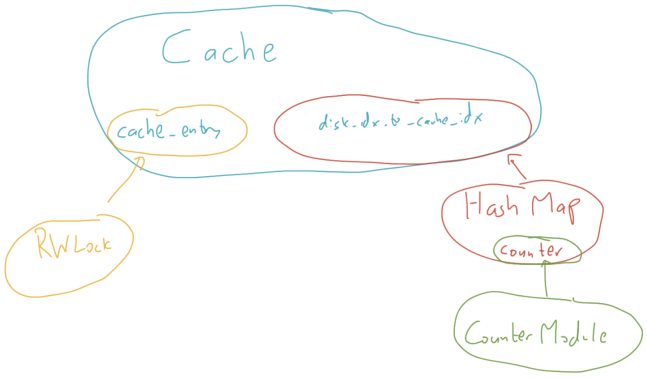
- The "base language" may provide some "points to" assertion $\ell \hookrightarrow v$, and then the user wants to build something else on top of that (like a fractional permission logic or similar) without needing to inject it into the native definition.
- You have some PCM N (custom built for your application), 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 PCM 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.



PCM 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 $\text{rel} : M \times N \to \mathit{bool}$. Of course, it must satisfy some conditions (and this is why we allow for flexibility in the definition of $\mapsto$ above).

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

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 PCM ghost state, which are mostly standard except the ones involving $\twoheadrightarrow$.

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

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

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

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

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

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

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

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; $\mathrm{active}(\kappa)$ indicates that lifetime κ is active, and $\mathrm{borrowed}(\kappa, A)$ 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?)

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

$$\frac{}{\text{active}(\kappa) * \text{borrowed}(\kappa, A) \Rrightarrow A} \text{ (BORROW-END)}$$

$$\frac{}{\&^\kappa A \Rrightarrow \&^\kappa A * \&^\kappa A} \text{ (BORROW-DUPE)}$$

$$\frac{\kappa_1 \sqsupseteq \kappa_2}{\&^{\kappa_1} A \Rrightarrow \&^{\kappa_2} A} \text{ (REBORROW)}$$

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

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

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

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

$$\frac{\forall r.\ m \le r \land n \le r \implies k \le r}{\&^\kappa \ulcorner m \urcorner^\gamma * \&^\kappa \ulcorner n \urcorner^\gamma \Rrightarrow \&^\kappa \ulcorner k \urcorner^\gamma} \text{ (BORROW-COMBINE)}$$

$$\frac{c \cdot a \mapsto c \cdot b}{\text{active}(\kappa) * \&^\kappa \ulcorner c \urcorner^\gamma * \ulcorner a \urcorner^\gamma \Rrightarrow \text{active}(\kappa) * \ulcorner b \urcorner^\gamma} \text{ (BORROW-GHOST-VIEWSHIFT)}$$

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

$$\frac{\begin{array}{c}\widehat{M} \twoheadrightarrow_{\text{rel}} \widehat{N} \\ \forall p, b.\ \text{rel}(m \cdot p, b) \implies a \le b\end{array}}{\&^\kappa \ulcorner m \urcorner^{\delta \twoheadrightarrow \gamma} \Rrightarrow \&^\kappa \ulcorner a \urcorner^\gamma} \text{ (BORROW-BACK)}$$

$$\frac{\begin{array}{c}\widehat{M} \twoheadrightarrow_{\text{rel}} \widehat{N} \\ \forall p, q.\ \text{rel}(c \cdot b \cdot p, q) \implies \exists q'.\ \text{rel}(c \cdot b' \cdot p, q') \land (a \cdot q \mapsto a' \cdot q')\end{array}}{\text{active}(\kappa) * \&^\kappa \ulcorner c \urcorner^{\delta \twoheadrightarrow \gamma} * \ulcorner b \urcorner^{\delta \twoheadrightarrow \gamma} * \ulcorner a \urcorner^\gamma \Rrightarrow \text{active}(\kappa) * \ulcorner b' \urcorner^{\delta \twoheadrightarrow \gamma} * \ulcorner a' \urcorner^\gamma} \text{ (BORROW-GHOST-REFINEMENT-EXCHANGE)}$$

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"

Justification

As before, our justification will be to roll all the ghost state, the $\&^\kappa \ulcorner a \urcorner^\gamma$ 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 $\text{active}(\kappa) * \&^\kappa \ulcorner a \urcorner^\gamma$ with $a \in M$. Then, consider:

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

- All state in an extension δ of γ, both *borrowed and non-borrowed*. By "extension" δ, we mean any location $\alpha \twoheadrightarrow \gamma$, $\alpha \twoheadrightarrow (\beta \twoheadrightarrow \gamma)$, and so on, (and *not* including γ itself). Project all this state down to $\boldsymbol{M}$ 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 $\&^{\kappa}\ulcorner a \urcorner^{\gamma}$, 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 PCM refinement be so effective for dealing with borrowed state.

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

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

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