Using Iris for Program Verification

Abel Nieto

May 4, 2020

Contents

1 Monotone Counter

▷ **Coq code:** `https://github.com/abeln/iris-practice/blob/master/counter.v` ◁

This is the counter example from Section 7.7 of the notes. The module exports three methods:

- `newCounter` creates a new counter. Counters are represented simply with an int reference.

  ```
  Definition newCounter : val := \lambda: <>, ref #0.
  ```

- `read` returns the value currently stored in a counter.

  ```
  Definition read : val := \lambda: "c", !"c".
  ```

- `incr` takes a counter, increments it, and returns unit.

  ```
  Definition incr : val :=
    rec: "incr" "c" :=
      let: "n" := !"c" in
      let: "m" := #1 + "n" in
      if: CAS "c" "n" "m" then #() else "incr" "c".
  ```

The client for the counter is a program that instantiates a counter, spawns two threads each incrementing the counter, and then reads the value off the counter:

```
Definition client : val :=
 \lambda: <>,
   let: "c" := newCounter #() in
   ((incr "c") ||| (incr "c")) ;;
   read "c".
```

1.1 Specs

We can prove two different specs for the code above:

- Authoritative RA

  In the first spec, our resource algebra is $\textsc{Auth}(\mathbb{N})$. Elements of this RA are of the form either $\bullet n$ (authoritative) or $\circ n$ (non-authoritative), where $n \in \mathbb{N}$.

  The important properties of this RA (for the counter example) are:

  - $\bullet 0 \cdot \circ 0 \in \mathcal{V}$
  - $\bullet m \cdot \circ n \in \mathcal{V}$ implies $m \geq n$
  - $\bullet m \cdot \circ n \rightsquigarrow \bullet(m+1) \cdot \circ(n+1)$
  - $| \circ n| = \circ n$

Then we can define our predicate for the counter:

$$\text{isCounter}(l, n, \gamma) = \boxed{\circ n}^{\gamma} * \exists s. \boxed{\exists m. l \hookrightarrow m \land \boxed{\bullet m}^{\gamma}}^{s}$$

This predicate is persistent because the invariant component is persistent and $\circ n$ is also persistent (via the PERSISTENTLY-CORE rule), because the core of $\circ n$ is $\circ n$ itself .

The intuition behind the predicate is that if we own $\text{isCounter}(l, n, \gamma)$ then, at an atomic step (e.g. a load), we can know that l is allocated and it points to a value that's $\geq n$.

Additionally, after an increment, we can update the resource to $\text{isCounter}(l, n+1, \gamma)$.

We then prove the following specs for the counter methods:

- $\{\top\}$ newCounter () $\{l.\exists\gamma.\text{isCounter}(l, 0, \gamma)\}$
- $\{\text{isCounter}(l, n, \gamma)\}$ read l $\{v.v \geq n\}$. Notice how we don't have to give the counter back, because it's persistent.
- $\{\text{isCounter}(l, n, \gamma)\}$ incr l $\{\text{isCounter}(l, n+1, \gamma)\}$

We can also prove the client spec:

$$\{\top\}\text{client }()\{n.n \geq 1\}$$

The "problem" with this specification *isCounter* doesn't tell us anything about what other threads are doing with the counter. So in the verification of the client code, after both threads return, we know

$$\text{isCounter}(l, 1, \gamma) * \text{isCounter}(l, 1, \gamma) \equiv \text{isCounter}(l, 1, \gamma)$$

So we know that the counter must be ≥ 1, but we don't know that it's 2.

- Authoritative RA + Fractions

  The second spec we can give is more precise, and allows us to conclude that the client reads exactly 2 once the two threads finish.

  The resource algebra we use is $\textsc{Auth}((\mathbb{Q}_p \times \mathbb{N})_?)$:

  - $\mathbb{Q}_p$ is the RA of positive fractions, where the valid elements are those ≤ 1.
  - The ? is the *optional RA* construction. In this case, it's needed because to use the authoritative RA we need the argument algebra to be *unital*. And $\mathbb{Q}_p \times \mathbb{N}$ is not unital because 0 is not an element of $\mathbb{Q}_p$.

  The important properties of this RA are:

  - $\circ(1,0) \cdot \bullet(1,0) \in \mathcal{V}$
  - $\circ(p,n) \cdot \bullet(1,m) \in \mathcal{V}$ implies $m \geq n$
  - $\circ(1,n) \cdot \bullet(1,m) \in \mathcal{V}$ implies $m = n$. This is the rule that will allows us to give a more precise specification.
  - $\circ(p,n) \cdot \bullet(q,m) \rightsquigarrow \circ(p,n+1) \cdot \bullet(q,m+1)$

With this RA, we get the new counter resource

$$\text{isCounter}(l,n,p,\gamma) = \boxed{\circ(p,n)}^{\gamma} * \exists s. \boxed{\exists m. l \hookrightarrow m \wedge \boxed{\bullet(1,m)}^{\gamma}}^{s}$$

This resource is no longer persistent, because the core $|p|$ is undefined. However, we can show the following

$$\text{isCounter}(l,n+m,p+q,\gamma) \dashv\vdash \text{isCounter}(l,n,p,\gamma) * \text{isCounter}(l,m,q,\gamma)$$

We then prove the following specs for the counter methods:

- $\{\top\}$ newCounter () $\{l.\exists\gamma.\text{isCounter}(l,0,1,\gamma)\}$
- $\{\text{isCounter}(l,n,p,\gamma)\}$ read l $\{v.v \geq n * \text{isCounter}(l,n,p,)\}$. Notice how we need to give back the counter in this case.
- $\{\text{isCounter}(l,n,1,\gamma)\}$ read l $\{v.v = n * \text{isCounter}(l,n,p,)\}$. Notice that since we have the entire fraction, we can get the exact value of the counter.
- $\{\text{isCounter}(l,n,p,\gamma)\}$ incr l $\{\text{isCounter}(l,n+1,p,\gamma)\}$

With this, we get the more precise client spec

$$\{\top\}\text{client }()\{n.n = 2\}$$

2 Locks and Coarse-Grained Bags

▷ **Coq code:** `https://github.com/abeln/iris-practice/blob/master/lock.v` ◁

This is the spin lock example from Section 7.6 of the notes.

The spin lock is a module with three methods:

- `newlock` creates a new lock. Locks are represented as a reference to a boolean. If `false`, the lock is free; if `true`, then it's taken.
- `acquire` uses a CAS cycle to spin until it can acquire the lock.
- `release` just sets the lock to `false`.

The code is as above:

```
Definition newlock : val := \lam: <>, ref #false.

Definition acquire : val :=
  rec: "acquire" "l" :=
    if: CAS "l" #false #true then #() else "acquire" "l".

Definition release : val := \lam: "l", "l" <- #false.
```

2.1 Spec

The involved RA is $\textsc{Ex}(\textsc{Unit})$.

- Notice that unlike other RA constructions, $\textsc{Ex}(S)$ just requires S to be a *set*, as opposed to another RA.
- The exclusive RA is defined by adding an element $\bot$ to the carrier set. For any two elements $x, y \neq \bot$, we have $x \cdot y = \bot$. Every element except for $\bot$ is valid.
- What this means is that if we own $\ulcorner x \urcorner^{\gamma}$, no other thread can own another $\ulcorner y \urcorner^{\gamma}$, because their product would be invalid.

The lock predicate is then

$$\text{isLock}(l, P, \gamma) = \exists s \boxed{l \hookrightarrow \texttt{false} \wedge P \wedge \ulcorner \texttt{()} \urcorner^{\gamma} \vee l \hookrightarrow \texttt{true}}^{s}$$

A first observation is that the lock protects an *arbitrary predicate* P, which is what the client uses to prove its correctness.

The intuition is the following:

- If the lock is *unlocked* ($l \hookrightarrow \texttt{false}$), then we need to *give away* to the invariant the predicate P protected by the lock *and* the key `()`. Because $\texttt{()} \in \textsc{Ex}(\textsc{Unit})$, we *know that no other key can be created.*
- When the lock is *locked* ($l \hookrightarrow \texttt{true}$), we don't need to give away any resources.
- When the lock transitions from unlocked to locked, we *gain* resources.
- When the lock transitions from locked to unlocked, we *lose* resources.

The last two points are reflected in the specs for the lock methods:

- $\{P\}\texttt{newlock}\ ()\{l.\exists\gamma.\text{isLock}(l, P, \gamma)\}$.

  Notice that even though only one element of $\textsc{Ex}(\textsc{Unit})$ is allowed, this restriction is *per location/ghost name*, so we're allowed to allocate a new (exclusive) ghost resource, proviced we furnish a new ghost name γ.

- $\{\text{isLock}(l, P, \gamma)\}$`acquire` $l\{P * \boxed{()}^{\gamma}\}$

  After acquiring the lock, we gain ownership of the predicate P and the key `()`.

- $\{\text{isLock}(l, P, \gamma) * P * \boxed{()}^{\gamma}\}$`release` $l\{\top\}$

  To release the lock, we need to show that P continues to hold, and that we have the key $\boxed{()}^{\gamma}$.

Also note that the lock predicate is *persistent*, since it's protected inside an invariant. This is important because multiple threads need to know that a given lock exists (so they can synchronize).

2.2 Client: Coarsed-Grained Bags

This client is also from the notes, and it consists of a *bag* data structure. We can create new bags, add items to it, and remove (the first) items from it.

▷ **The implementation of the bag looks just like a stack. But the specification says nothing about the order of insertions/removals, that's why it's a bag and not a stack** ◁.

The bag code follows:

```
Section bag_code.

  (* A bag is a pair of (optional list of values, lock)  *)
  Definition new_bag : val :=
    \lam: <>, (newlock #(), ref NONEV).

  (* Insert a value into the bag. Return unit. *)
  Definition insert : val :=
    \lam: "bag" "v",
      let: "lst" := Snd "bag" in
      let: "lock" := Fst "bag" in
      acquire "lock";;
      "lst" <- SOME ("v", !"lst");;
      release "lock";;
      #().

  (* Remove the value last-added to the bag (wrapped in an option), if the
     bag is non-empty. If the bag is empty, return NONE. *)
  Definition remove : val :=
    \lam: "bag",
      let: "lst" := Snd "bag" in
      let: "lock" := Fst "bag" in
      acquire "lock";;
      let: "res" :=
         match: !"lst" with
```

```
           NONE => NONEV
         | SOME "pair" =>
           "lst" <- Snd "pair";;
           SOME (Fst "pair")
         end
      in
      release "lock";;
      "res".

End bag_code.
```

2.2.1 Bag Specification

The spec we want to show says only elements satisfying some predicate Φ can be added to the bag. In turn, we guarantee that any element x removed from the bag satisfies Φx.

The bag predicate looks like

$$\text{isBag}(b, \Phi, \gamma) = \exists l, lst. b = (lock, lst) \wedge \text{isLock}(l, \exists n, vs. lst \hookrightarrow vs \wedge \text{baglist}(vs, n, \Phi), \gamma)$$

In turn, $\text{baglist}(vs, n, \Phi)$ is a predicate that expresses that vs is a HeapLang list of n elements, all of which satisfy Φ.

$$\begin{aligned} \text{baglist}(vs, n, \Phi) = & (n = 0 \wedge vs = \texttt{None}) \\ & \vee (n = Sn' \wedge \\ & \exists v, vs'. vs = \texttt{Some}((v, vs')) \wedge \Phi v \wedge \text{baglist}(vs, n', \Phi)) \end{aligned}$$

▷ **The notes use guarded recursion to define baglist. I ended using the index n. Need to try the guarded recursion approach.** ◁

The specs for the different methods are:

- $\{T\}\texttt{newbag ()}\{v.\gamma\text{isBag}(v, \Phi, \gamma)\}$
- $\{\Phi v * \text{isBag}(b, \Phi, \gamma)\}\texttt{insert } b\ v\{T\}$
- $\{\text{isBag}(b, \Phi, \gamma)\}\texttt{remove b}\{v.v = \texttt{None} \vee \exists x.v = \texttt{Some}(x) * \Phi x\}$

Notice that `isBag` is also persistent/duplicable.

▷ **Need to implement concurrent client for bags** ◁

3 Message-Passing Idiom

▷ **Coq code:** `https://github.com/abeln/iris-practice/blob/master/weakmem.v` ◁

This is the “message passing” example from the “Strong Logic for Weak Memory’ paper’: `https://people.mpi-sws.org/~dreyer/papers/iris-weak/paper.pdf`.

3.1 Code

The program is simple: we have two variables x and y that start as 0, and are then mutated by two threads. The second thread loops until y is non-zero, which restricts the order of execution:

```
(* First we have a function ‘repeat l‘, which reads l until its value is non-zero,
   at which point it returns l’s value. *)
Definition repeat_prog : val :=
  rec: "repeat" "l" :=
    let: "vl" := !"l" in
    if: "vl" = #0 then ("repeat" "l") else "vl".

(* Then we have the code for the example. *)
Definition mp : val :=
  \lambda: <>,
     let: "x" := ref #0 in
     let: "y" := ref #0 in
     let: "res" := (("x" <- #37;; "y" <- #1) ||| (repeat_prog "y";; !"x")) in
     Snd "res".
```

3.2 Specification

This example uses impredicative invariants[1]: specifically one invariant for x that's embedded inside another invariant that talks about y (notice we're interested in the value of x at the end):

- The (inner) invariant for x is $Inv_x \triangleq \boxed{x \hookrightarrow 37 \lor \ulcorner () \urcorner^{\gamma}}^{l_x}$.

- The (outer) invariant for y is $Inv_y \triangleq \boxed{y \hookrightarrow 0 \lor y \hookrightarrow 1 \land Inv_x}^{l_y}$

  Notice we use the Ex(Unit) RA.
  How does the verification work?

- We'll give the left thread the resources $Inv_y * x \hookrightarrow 0$. The first update of x can be done without problem. The second update for y forces us to "choose" the second term in the disjunction. We have to give up ownership of x to the invariant.

- The right thread gets the resources $Inv_y * \ulcorner () \urcorner^{\gamma}$. The proof is by Lob induction (since the function is recursive). There are two cases to consider, induced by the test of y's value in `repeat`.

  – If $y = 0$, then we use the induction hypothesis.

[1] Leon points out that we don't really need impredicativity for this example: we might as well have inlined the inner invariant.

- If $y = 1$, then we know we're in the second case of the invariant and, further, that $x \hookrightarrow 37$ (because the resource is exclusive). In this case, we're able to "swap" $x \hookrightarrow 37$ for $\boxed{()}^{\gamma}$ when we close the invariant, leaving us with ownership of $x \hookrightarrow 37$ (although this is not required to verify the example).

3.3 Getting Ownership of both x and y

I also did a variant of the exercise (suggested by Leon) where we want to end up with ownership of *both* variables.

I ended doing this in quite a "mechanistic" way, which suggests possibilities for automation (this had also been suggested by Leon).

The idea is to define an invariant that lists all the possible "actual" values of x and y:

$$Inv \triangleq S_1 \lor S_2 \lor S_3 \lor S_4$$

Each S_i contains the state of the heap plus a "key" that's needed to "access" the state.

$$\begin{aligned}
S_1 &= y \hookrightarrow 0 * x \hookrightarrow 0 * \boxed{()}^{\gamma_2} * \boxed{()}^{\gamma_3} * \boxed{()}^{\gamma_4} \\
S_2 &= y \hookrightarrow 0 * x \hookrightarrow 37 * \boxed{()}^{\gamma_1} * \boxed{()}^{\gamma_3} * \boxed{()}^{\gamma_4} \\
S_3 &= y \hookrightarrow 1 * x \hookrightarrow 37 * \boxed{()}^{\gamma_1} * \boxed{()}^{\gamma_2} * \boxed{()}^{\gamma_4} \\
S_4 &= \boxed{()}^{t_1} * \boxed{()}^{t_2} * \boxed{()}^{\gamma_1} * \boxed{()}^{\gamma_2} * \boxed{()}^{\gamma_3}
\end{aligned}$$

▷ **explain in more detail** ◁

3.4 ▷ Questions ◁

- Can we open invariants at "dummy atomic statements"? We might want to do this to "exchange" some tokens by others.

4 Bags with Helping

▷ **Coq code:** `https://github.com/abeln/iris-practice/blob/master/helping.v` ◁

This is a larger example, also from the lecture notes. The idea of this example is to implement a stack with an optimization that consists of a "mailbox". This mailbox is just a memory location where threads can temporarily please and remove items so as to avoid having to push and pop in certain cases.

Specifically, the mailbox comes in handy in the following situation:

- Suppose we have two threads A and B, and the stack is currently empty.

- Thread A wants to push an element to the stack.
- Thread B wants to remove an element.
- Without mailboxes, B would wait until A pushes, then B would pop.
- With the mailbox, A instead of pushing places its element in the mailbox. B notices it and then removes the element. No pushes or pops were necessary.
- ▷ **Question: how much time/instructions are we actually saving with this?** ◁

4.1 Offers

The mailbox is itself implemented in terms of a lower-level abstraction: an "offer".

An offer is a pair (v, l), where v is the value in the offer and l is a location indicating the offer status:

- $l \hookrightarrow 0$ is the initial state of the offer
- $l \hookrightarrow 1$ means the offer has been accepted
- $l \hookrightarrow 2$ means the offer has been revoked (an offer can only be revoked by the thread that created it)

Offer Code

```
Definition mk_offer : val := \lam: "v", ("v", ref #0).

Definition revoke_offer : val :=
  \lam: "off",
    let: "v" := Fst "off" in
    let: "l" := Snd "off" in
    if: (CAS "l" #0 #2) then SOME "v" else NONE.

Definition accept_offer : val :=
  \lam: "off",
    let: "v" := Fst "off" in
    let: "l" := Snd "off" in
    if: (CAS "l" #0 #1) then SOME "v" else NONE.
```

Offer Specification

The representation predicate for offers uses the state transition "trick" to encode the three states of the offer, plus the fact that only the creator can revoke an offer.

$$\text{isOffer}(o)_\gamma \triangleq \exists v, l. o = (v, l) * \boxed{l \hookrightarrow 0 * \Phi v \lor l \hookrightarrow 1 \lor (l \hookrightarrow 2) * \boxed{()}^{\gamma}}^{l}$$

- Notice this is duplicable, as usual.
- Notice the predicate is indexed by γ, so an offer can only be revoked by a thread holding $\boxed{()}^{\gamma}$, the “key”.
- If $l \hookrightarrow 0$, we also know Φv. This ensures that every element in the bag satisfies the requisite predicate Φ. This is as per the bag spec in a previous example.

The method specs are

- $\{\Phi v\}$`mk_offer v`$\{o.\exists\gamma.\text{isOffer}(o)_{\gamma}\}$
- $\{\text{isOffer}(o)_{\gamma}\}$`accept_offer o`$\{v.v = \mathit{None} \lor \exists w.v = \mathit{Some}(w) * \Phi w\}$
- $\{\text{isOffer}(o)_{\gamma} * \boxed{()}^{\gamma}\}$`revoke_offer o`$\{v.v = \mathit{None} \lor \exists w.v = \mathit{Some}(w) * \Phi w\}$. **▷ The specs of accept and revoke are identical, except we require ownership of the key for revoke. ◁**

4.2 Mailboxes

Offers are values, so they can't be mutated (well, the location component of an offer, which indicates state, can be mutated). A mailbox is then a location pointing to an offer.

Following the lecture notes, we implement mailboxes as a location, plus two closures over that location:

```
Definition mk_mailbox : val :=
  \lam: <>,
     let: "r" := ref NONEV in

     let: "put" :=
       (\lam: "v",
         let: "o" := mk_offer "v" in
         "r" <- SOME "o";;
         revoke_offer "o")
     in

     let: "get" :=
       (\lam: "v",
         match: !"r" with
           NONE => NONEV
         | SOME "o" => accept_offer "o"
         end)
     in

     ("put", "get").
```

Comments:

- **▷ What are the pros/cons of this style with closures preferred over just declaring each function at the top level? ◁**
- It's interesting that `put` creates an offer and "right away" revokes it. Of course, since `put` and `get` are not synchronized, there's room for a concurrent execution of `get` to remove the item in the offer (and the notes do mention that in a realistic setting we'd want a timeout here).
- **▷ Is there a version of separation logic for reasoning about time? e.g. reasoning about programs with real-time constraints? Would we run into liveness issues? ◁**
- See "separation logic with time credits" for doing amortized complexity analysis: `https://link.springer.com/article/10.1007/s10817-017-9431-7`.

The representation predicate for mailboxes is straightforward:

$$\text{isMailbox}(l) \triangleq= l \hookrightarrow \mathit{None} \vee \exists o, \gamma . l \hookrightarrow \mathit{Some}(o) * \text{isOffer}(o)_\gamma$$

Within the proof of the spec (I haven't shown the spec yet), we need to box the above in an invariant.

The spec for `mk_mailbox` is what we would expect (both `get` and `put` return either a None or $\mathit{Some}(v)$, for which we know Φv). **▷ The one interesting thing is this spec uses nested Hoare triples. ◁**

4.3 Stack

Finally we get to the stack implementation and its bag specification. Again we use the same closure style. See Figure 1.

A lot of the complexity in the specification comes from the fact that, unlike in the lecture notes, the Coq mechanization of Iris cannot do a `CAS` on arbitrary values, only on *unboxed* ones. In particular, we can compare locations, but not *pairs*, with a `CAS`.

The specifications of push and pop are the regular bag specifications we've already seen.

The tricky thing is coming up with the invariant for the stack. I stole the idea from the solution in the Iris repo. However, the solution in the repo uses guarded recursion, which I didn't use, so I had to adapt things.

First, the stack invariant:

$$\text{stackInv}(l) \triangleq \exists o, ls . l \hookrightarrow \texttt{oloc\_to\_value o} * \text{isStack}(o, ls)$$

- o here is an *Option* that wraps a *location*. What we really want to say is that either l points to None or to $\mathit{Some}(l')$, but if we say it like that then we introduce duplication in the proofs. **▷ This factoring of useful information in invariants seems important. ◁**

```
Definition mk_stack : val :=
   \lam: <>,
      let: "mb" := mk_mailbox #() in
      let: "put" := Fst "mb" in
      let: "get" := Snd "mb" in

      (* The stack is a pointer p that points to either
         - NONEV, if the stack is empty
         - SOMEV l, if the stack is non-empty.
           l in turn points to a pair (h, t), where h is the top of the stack
           and ‘t‘ is the rest of the stack. *)

      let: "stack" := ref NONEV in

      (* Push an element into the stack. Returns unit. *)
      let: "push" :=
         rec: "push" "v" :=
           match: ("put" "v") with
             NONE => #()
           | SOME "v" =>
             let: "curr" := !"stack" in
             let: "nstack" := SOME (ref ("v", "curr")) in
             if: (CAS "stack" "curr" "nstack") then #() else ("push" "v")
           end
      in

      (* Pop an element from the stack. If the stack is empty, return None,
         otherwise return Some head. *)
      let: "pop" :=
       rec: "pop" <> :=
         match: ("get" #()) with
           SOME "v" => SOME "v"
         | NONE =>
           match: !"stack" with
             NONE => NONEV
           | SOME "l" =>
             let: "p" := !"l" in
             let: "head" := Fst "p" in
             let: "tail" := Snd "p" in
             if: (CAS "stack" (SOME "l") "tail") then SOME "head" else ("pop" #())
           end
         end
      in

      ("push", "pop").
```

Figure 1: Stack implementation with mailboxes

- The oloc_to_value helper is defined thus

  ```
  Definition oloc_to_val (o : option loc) : val :=
    match o with
      None => NONEV
    | Some l => SOMEV #l
    end.
  ```

- The representation predicate for stacks is generated by the function

  ```
  Fixpoint is_stack (o : option loc) (ls : list val) : iProp :=
    match ls with
      nil => o = None
    | x :: xs => \exists (l : loc) (o' : option loc),
      o = Some l * l ->{-}(x, oloc_to_val o') * \Phi x * (is_stack o' xs)
  end.
  ```

  ▷ TODO: find a way to phrase the predicate without using the option trick ◁

  Notice here we have the fractional points-to predicate. **▷ This means that when we open the invariant we'll get write access to the first element of the list, but only read access to the rest of the elements. This is crazy! ◁**

- We can then prove the following crucial lemma

  ```
  Lemma is_stack_readonly_dup o ls :
  is_stack o ls -* is_stack o ls *
                           (match ls with
                              nil => True
                            | x :: xs =>  \exists (l : loc) (o' : option loc),
                              o = Some l * l ->{-} (x, oloc_to_val o')
                            end)
  ```

  This lemma is useful in the following step

  ```
  match: !"stack" with
                NONE => NONEV
              | SOME "l" =>
                let: "p" := !"l" in
                let: "head" := Fst "p" in
  ```

  We open the invariant, read stack, and know that it points to a pointer l that points to a pair. However, we need to close the invariant, but we *also* need to preserve the knowledge about the value that l points to. This is precisely what is_stack_readonly_dup gives us.

  The other important lemma is mapsto_agree

```
mapsto_agree
     : \forall (l : ?L) (q1 q3 : Qp) (v2 v3 : ?V),
     mapsto l q1 v2 -* mapsto l q3 v3 -* v2 = v3
```

This lemma allows us to re-assert full ownership of the head of the stack after the CAS, so we can re-establish the invariant.

5 Modular Specifications

Lemma 1. $\gamma \Mapsto^q n * \gamma \Mapsto^q m \vdash n = m$

Proof. We have $\gamma \Mapsto^q n = \boxed{(q,n)}^{\gamma}$ and $\gamma \Mapsto^q m = \boxed{(q,m)}^{\gamma}$. Then

$$\begin{aligned}
\boxed{(q,n)}^{\gamma} * \boxed{(q,m)}^{\gamma} &\vdash \boxed{(q,n)\cdot(q,m)}^{\gamma} && \text{Own-op}\\
&\vdash (q,n)\cdot(q,m) \in V && \text{Own-valid}\\
&\vdash (2q, n\cdot m) \in V\\
&\vdash n\cdot m \in V\\
&\vdash n = m
\end{aligned}$$

□

Lemma 2. $\gamma \Mapsto^p m * \gamma \Mapsto^q m \dashv\vdash \gamma \Mapsto^{p+q} m$

Proof. Follows directly from the properties of the resource algebra $\mathbb{Q}_p \times \textsc{Agree}(\mathbb{N})$. □

Lemma 3. $\gamma \Mapsto^1 m \vdash (\mathrel{|\!\!\Rrightarrow} \gamma \Mapsto^1 n)$

Proof. We use Ghost-update and then have to show $(1,m) \rightsquigarrow (1,n)$. That, suppose that $(1,m)\cdot(q,m') \in V$. Then we have to show that $(1,n)\cdot(q,m') \in V$.

Notice that $(1,m)\cdot(q,m') \in V$ implies $1 \cdot q \in V$. In turn, this implies $1 + q \in V$, which means $1 + q \leq 1$. But this can't be, because we know $q > 0$. This means that there's no such (q,m') and the result follows immediately. □

5.1 Modular Counter Spec

The modular counter spec is

$$\begin{aligned}
&\exists Cnt : Val \to GhostName \to InvName \to Prop.\\
&\qquad \Box(\forall v, \gamma, c. Cnt(v,\gamma,c) \implies \Box Cnt(v,\gamma,c))\\[1em]
&\qquad \wedge \{True\} newCounter(())\{v.\exists\gamma, c. Cnt(v,\gamma,c) * \gamma \Mapsto^{\frac{1}{2}} 0\}_{\varepsilon}\\[1em]
&\qquad \wedge \forall v, \gamma, c, P, Q.(\forall m. \gamma \Mapsto^{\frac{1}{2}} m * P \Rrightarrow_{\varepsilon\setminus\{c\}} \gamma \Mapsto^{\frac{1}{2}} m * Q(m)) \implies\\
&\qquad \{Cnt(v,\gamma,c) * P\} read(v)\{w. Cnt(v,\gamma,c) * Q(w)\}_{\varepsilon}\\[1em]
&\qquad \wedge \forall v, \gamma, cP, Q.(\forall m. \gamma \Mapsto^{\frac{1}{2}} m * P \Rrightarrow_{\varepsilon\setminus\{c\}} \gamma \Mapsto^{\frac{1}{2}} m + 1 * Q(m)) \implies\\
&\qquad \{Cnt(v,\gamma,c) * P\} incr(v)\{w. Cnt(v,\gamma,c) * Q(w)\}_{\varepsilon}\\[1em]
&\qquad \wedge \forall v, \gamma, c, m, q.(\gamma \Mapsto^{\frac{1}{2}} m * P \Rrightarrow_{\varepsilon\setminus\{c\}} \gamma \Mapsto^{\frac{1}{2}} m + 1 * \gamma \Mapsto^{q} m + 1 * Q) \implies\\
&\qquad \{Cnt(v,\gamma,c) * \gamma \Mapsto^{q} m * P\} wk\_incr(v)\{w. w = () * Cnt(v,\gamma,c) * \gamma \Mapsto^{q} m + 1 * Q\}_{\varepsilon}
\end{aligned}$$

5.1.1 Realizing the Spec

We instantiate the abstract predicate by

$$Cnt(v,\gamma,c) = \boxed{\exists m. v \hookrightarrow m * \gamma \Mapsto^{\frac{1}{2}} m}^{c}$$

We have four proof obligations to realize the spec:

1. Cnt is persistent: $Cnt(v,\gamma,c) \vdash \Box Cnt(v,\gamma,c)$

   Proof. Follows by Inv-persistent. □

2. $\vdash \{True\} newCounter(())\{v.\exists\gamma, c. Cnt(v,\gamma,c) * \gamma \Mapsto^{\frac{1}{2}} 0\}$

   Proof. $newCounter$ just allocates a location pointing to 0, so after the allocation we know $v \hookrightarrow 0$. We then show the view shift

   $$\vdash v \hookrightarrow 0 \Rrightarrow \exists\gamma. v \hookrightarrow 0 * \gamma \Mapsto^{1} 0$$

   Two rules come in handy for the above:

   $$\frac{\vdash P \implies |\!\Rrightarrow Q}{\vdash P \Rrightarrow Q}$$

   and $P * |\!\Rrightarrow Q \vdash |\!\Rrightarrow (P * Q)$

   We can then break up the abstract ownership to get

$$\vdash v \hookrightarrow 0 \Rrightarrow \exists \gamma.(v \hookrightarrow 0 * \gamma \Mapsto^{\frac{1}{2}} 0) * \gamma \Mapsto^{\frac{1}{2}} 0$$

We can then allocate the invariant. I'm not sure which rule to use to allocate the invariant at this point.

□

3. (**read**)

$$(\forall m. \gamma \Mapsto^{\frac{1}{2}} m * P \Rrightarrow \gamma \Mapsto^{\frac{1}{2}} m * Q(m)) \implies \\ \{Cnt(v, \gamma, c) * P\} read(v) \{w. Cnt(v, \gamma, c) * Q(w)\}$$

We get to assume $\forall m. \gamma \Mapsto^{\frac{1}{2}} m * P \Rrightarrow \gamma \Mapsto^{\frac{1}{2}} m * Q(m)$ and need to show

$$\{Cnt(v, \gamma, c) * P\} read(v) \{w. Cnt(v, \gamma, c) * Q(w)\}$$

- We know that $read(v)$ is just $!v$.
- To read v, we open the invariant, and get $\triangleright \exists m. v \hookrightarrow m * \gamma \Mapsto^{\frac{1}{2}} m$.
- After reading v, we know that the result is some m and also know that $v \hookrightarrow m * \gamma \Mapsto^{\frac{1}{2}} m$. We also know P from the hypothesis.
- We can then instantiate our view shift hypothesis to conclude $\gamma \Mapsto^{\frac{1}{2}} m * Q(m)$.
- We can close the invariant because the physical and abstract state of the counter agree.

4. (**incr**)

$$\forall v, \gamma, cP, Q. (\forall m. \gamma \Mapsto^{\frac{1}{2}} m * P \Rrightarrow_{\varepsilon \setminus \{c\}} \gamma \Mapsto^{\frac{1}{2}} m + 1 * Q(v)) \implies \\ \{Cnt(v, \gamma, c) * P\} incr(v) \{w. Cnt(v, \gamma, c) * Q(w)\}_{\varepsilon}$$

We have

```
incr(v) =
  let m = !v in
  let m' = m + 1 in
  if (CAS(v, m, m')) then m else incr(v)
```

We assume the view shift and then need to show

$$\{Cnt(v, \gamma, c) * P\} incr(v) \{w. Cnt(v, \gamma, c) * Q(w)\}_{\varepsilon}$$

- We proceed by Lob induction, since incr is recursive.

- To read $!v$, we open the invariant, do the load, and close the invariant. The result is that $m = n$, for some n.
- Then we set $m' = n + 1$.
- Now we're at the CAS, and we know $Cnt(v, \gamma, c) * P$. Again we open the invariant, and then we know $\exists n'.v \hookrightarrow n' * \gamma \Mapsto^{\frac{1}{2}} n' * P$. We consider two cases: $n' = n$ or $n' \neq n$.
  - If $n' \neq n$, then we recurse and can use the induction hypothesis.
  - If $n' = n$, then we know $v \hookrightarrow n * \gamma \Mapsto^{\frac{1}{2}} n * P$. At that point, we can use the view shift to conclude $\gamma \Mapsto^{\frac{1}{2}} n + 1 * Q(n)$. Because the CAS succeeds, we also know that $v \hookrightarrow n + 1$, so we can close the invariant.

5. (**wk_incr**)

$$\forall v, \gamma, c, m, q.(\gamma \Mapsto^{\frac{1}{2}} m * P \Rrightarrow_{\varepsilon \setminus \{c\}} \gamma \Mapsto^{\frac{1}{2}} m + 1 * \gamma \Mapsto^{q} m + 1 * Q) \implies$$
$$\{Cnt(v, \gamma, c) * \gamma \Mapsto^{q} m * P\} wk\_incr(v) \{w.w = () * Cnt(v, \gamma, c) * \gamma \Mapsto^{q} m + 1 * Q\}_{\varepsilon}$$

We have

$$wk\_incr(v) \triangleq v \leftarrow 1+!v$$

We assume

$$\gamma \Mapsto^{\frac{1}{2}} m * P \Rrightarrow_{\varepsilon \setminus \{c\}} \gamma \Mapsto^{\frac{1}{2}} m + 1 * \gamma \Mapsto^{q} m + 1 * Q$$

and need to show

$$\{Cnt(v, \gamma, c) * \gamma \Mapsto^{q} m * P\} wk\_incr(v) \{w.w = () * Cnt(v, \gamma, c) * \gamma \Mapsto^{q} m+1 * Q\}_{\varepsilon}$$

- We use the bind rule to focus on $!v$, and open the invariant. We get to assume

$$\triangleright(\exists m'.\gamma \hookrightarrow m' * \gamma \Mapsto^{\frac{1}{2}} m') * \gamma \Mapsto^{q} m * P$$

- Because of the agreement construction and the lemma we proved above, we'll get $m' = m$.
- After reading $!v$, we have $Cnt(v, \gamma, c) * \gamma \Mapsto^{q} m * P$.
- Then we need to do the write $v \leftarrow 1 + m$. Again we open the invariant, and again we can know the value of the pointer (m). We have $\gamma \Mapsto^{\frac{1}{2}} m * \gamma \Mapsto^{q} m * P * v \hookrightarrow m$.
- After the write, we know

$$v \hookrightarrow m + 1 * \gamma \Mapsto^{\frac{1}{2}} m * \gamma \Mapsto^{q} m * P$$

- We now use our assumption about the view shift to conclude

  $$\gamma \Mapsto^{\frac{1}{2}} m+1 * \gamma \Mapsto^{q} m+1 * Q$$

  So in total, we own

  $$v \hookrightarrow m+1 * \gamma \Mapsto^{\frac{1}{2}} m+1 * \gamma \Mapsto^{q} m+1 * Q$$

- We can close the invariant to get

  $$Cnt(v,\gamma,c) * \gamma \Mapsto^{q} m+1 * Q$$

  as needed.

- We didn't use the restriction that the view shift happens with the mask $\varepsilon \backslash \{c\}$.

- The lecture notes are missing $\gamma \Mapsto^{q} m+1$ in the postcondition.

- What if we didn't provide the $\gamma \Mapsto^{q} m$ resource? In that case, the Hoare triple would be

  $$\{Cnt(v,\gamma,c) * P\} wk\_incr(v) \{w.w = () * Cnt(v,\gamma,c) * Q\}_{\varepsilon}$$

  We wouldn't know the value of the counter when $wk\_incr(v)$ finishes. This makes sense, because other threads could be incrementing the counter at the same time.

5.1.2 Derived Sequential Spec

This is the desired sequential spec:

$$\begin{aligned} \exists SeqCnt : & Val \to GhostName \to InvName \to \mathbb{N} \to Prop. \\ & \wedge \{True\} newCounter(()) \{v. \exists \gamma, c. SeqCnt(v,\gamma,c,0)\} \\ & \wedge \{SeqCnt(v,\gamma,c,n)\} read(v) \{w.w = n * SeqCnt(v,\gamma,c,n)\} \\ & \wedge \{SeqCnt(v,\gamma,c,n)\} incr(v) \{w.w = n * SeqCnt(v,\gamma,c,n+1)\} \end{aligned}$$

Where we set

$$SeqCnt(v,\gamma,c,n) \triangleq Cnt(v,\gamma,c) * \gamma \Mapsto^{\frac{1}{2}} n$$

Proofs:

- (**newCounter**): We know

  $$\{True\}newCounter(())\{v.\exists\gamma, c.Cnt(v,\gamma,c) * \gamma \Mapsto^{\frac{1}{2}} 0\}_{\varepsilon}$$

  and need to derive

  $$\{True\}newCounter(())\{v.\exists\gamma, c.SeqCnt(v,\gamma,c,0)\}$$

  This follows immediately from the definition of $SeqCnt$.

- (**read**): We know

  $$\begin{gathered}(\forall m.\gamma \Mapsto^{\frac{1}{2}} m * P \Rrightarrow \gamma \Mapsto^{\frac{1}{2}} m * Q(m)) \implies \\ \{Cnt(v,\gamma,c) * P\}read(v)\{w.Cnt(v,\gamma,c) * Q(w)\}\end{gathered}$$

  and need to derive

  $$\{SeqCnt(v,\gamma,c,n)\}read(v)\{w.w = n * SeqCnt(v,\gamma,c,n)\}$$

  We set

  - $P = \gamma \Mapsto^{\frac{1}{2}} n$. This means that

    $$\begin{aligned}SeqCnt(v,\gamma,c,n) &= Cnt(v,\gamma,c) * \gamma \Mapsto^{\frac{1}{2}} n \\ &= Cnt(v,\gamma,c) * P\end{aligned}$$

  - $Q = \lambda w.(w = n * \gamma \Mapsto^{\frac{1}{2}} n)$. What we're doing here is instantiating P and Q to give us the resources we need in the pre and post conditions.
  - All that remains is to show

    $$\vdash \forall m.\gamma \Mapsto^{\frac{1}{2}} m * \gamma \Mapsto^{\frac{1}{2}} n \Rrightarrow \gamma \Mapsto^{\frac{1}{2}} m * (m = n * \gamma \Mapsto^{\frac{1}{2}} n)$$

    We use this rule to drop the persistent modality at the top level (obscured by the view shift):

    $$\frac{\vdash P \implies \mid\!\Rrightarrow Q}{\vdash P \Rrightarrow Q}$$

    Then we have to show

    $$\gamma \Mapsto^{\frac{1}{2}} m * \gamma \Mapsto^{\frac{1}{2}} n \implies \mid\!\Rrightarrow \gamma \Mapsto^{\frac{1}{2}} m * (m = n * \gamma \Mapsto^{\frac{1}{2}} n)$$

    The agreement construction gives us $m = n$, and then the result follows from $P \vdash \mid\!\Rrightarrow P$.

The $m = n$ part was pretty magical. It is important because it links the physical and logical states. In this case, it worked because we're using the agreement construction. Do we always use the agreement construction for these modular specs?

- (**incr**):

  We know

  $$\forall v, \gamma, c, P, Q. (\forall m. \gamma \Mapsto^{\frac{1}{2}} m * P \Rrightarrow_{\varepsilon \setminus \{c\}} \gamma \Mapsto^{\frac{1}{2}} m + 1 * Q(v)) \implies \\ \{Cnt(v, \gamma, c) * P\} incr(v) \{w. Cnt(v, \gamma, c) * Q(w)\}_{\varepsilon}$$

  And need to show

  $$\{SeqCnt(v, \gamma, c, n)\} incr(v) \{w. w = n * SeqCnt(v, \gamma, c, n + 1)\}$$

  We set

  - $P = \gamma \Mapsto^{\frac{1}{2}} n$
  - $Q = \lambda x. x = n * \gamma \Mapsto^{\frac{1}{2}} n + 1$ The notes are missing the $x = n$ part.

  The pre and post conditions match up, so we're left with showing

  $$\forall m. \gamma \Mapsto^{\frac{1}{2}} m * \gamma \Mapsto^{\frac{1}{2}} n \Rrightarrow_{\varepsilon \setminus \{c\}} \gamma \Mapsto^{\frac{1}{2}} m + 1 * m = n * \gamma \Mapsto^{\frac{1}{2}} n + 1$$

  - Again, from the agreement construction, we know that $m = n$, so we get

    $$\gamma \Mapsto^{\frac{1}{2}} n * \gamma \Mapsto^{\frac{1}{2}} n \implies \mathrel{|\!\Rrightarrow} (\gamma \Mapsto^{\frac{1}{2}} n + 1 * \gamma \Mapsto^{\frac{1}{2}} n + 1)$$

  Since we have full ownership of γ, we can perform the frame-preserving update.

- When things go wrong. What if we wanted to derive an incorrect spec? Say we want to show that *incr* returns $n + 1$ as opposed to n.

  $$\{SeqCnt(v, \gamma, c, n)\} incr(v) \{w. w = n + 1 * SeqCnt(v, \gamma, c, n + 1)\}$$

  Then we'd have to choose $Q = \lambda x. x = n + 1 * \gamma \Mapsto^{\frac{1}{2}} n + 1$.

  We'd have to show

  $$\forall m. \gamma \Mapsto^{\frac{1}{2}} m * \gamma \Mapsto^{\frac{1}{2}} n \Rrightarrow_{\varepsilon \setminus \{c\}} \gamma \Mapsto^{\frac{1}{2}} m + 1 * m = (n + 1) * \gamma \Mapsto^{\frac{1}{2}} n + 1$$

The agreement construction would give us $n = m = (n+1)$, a contradiction. That's good!

What if the implementation did return $n+1$?

Consider the point of the proof of the modular spec of *incr* where we've done the CAS and it was successful:

- If $n' = n$, then we know $v \hookrightarrow n * \gamma \Mapsto^{\frac{1}{2}} n * P$. At that point, we need to use the view shift.

  $$\forall m. \gamma \Mapsto^{\frac{1}{2}} m * P \Rrightarrow_{\varepsilon\setminus\{c\}} \gamma \Mapsto^{\frac{1}{2}} m+1 * Q(m)$$

  We only have one possible instantiation for the quantifier, which is $m = n$, so from the view shift we get $\gamma \Mapsto^{\frac{1}{2}} n{+}1{*}Q(n)$. Unfortunately (fortunately for the soundness of the logic), we need $Q(n+1)$ and not $Q(n)$, so the proof doesn't go through.

5.1.3 Derived Concurrent Spec

▷ **TODO** ◁

6 Modular Specification of Message Passing Example