Using Iris for Program Verification

Abel Nieto

April 16, 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:

  $$\text{isCounter}(l, n, \gamma) = \left[\!\left[\circ n\right]\!\right]^{\gamma} * \exists s. \boxed{\exists m. l \hookrightarrow m \wedge \left[\!\left[\bullet m\right]\!\right]^{\gamma}}^{s}$$

  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

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

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 $\perp$ to the carrier set. For any two elements $x, y \neq \perp$, we have $x \cdot y = \perp$. Every element except for $\perp$ is valid.
- What this means is that if we own $\boxed{x}^{\gamma}$, no other thread can own another $\boxed{y}^{\gamma}$, because their product would be invalid.

The lock predicate is then

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

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.

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

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

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

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

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:

- $\{T\}$`newbag` $()\{v.\gamma\text{isBag}(v, \Phi, \gamma)\}$
- $\{\Phi v * \text{isBag}(b, \Phi, \gamma)\}$`insert` $b\ v\{T\}$
- $\{\text{isBag}(b, \Phi, \gamma)\}$`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 $Inv_x \triangleq \boxed{x \hookrightarrow 37 \lor \left[\,()\,\right]^{\gamma}}^{l_x}$.
- 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 $Inv_y * \left[\,()\,\right]^{\gamma}$. 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.
  - 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 $\left[\,()\,\right]^{\gamma}$ when we close the invariant, leaving us with ownership of $x \hookrightarrow 37$ (although this is not required to verify the example).

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

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:

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

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 pushs or pops were necessary.

- **$\triangleright$ Question: how much time/instructions are we actually saving with this? $\triangleleft$**

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.

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

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

4.2 Mailboxes