Using Iris for Program Verification

Abel Nieto

March 27, 2020

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 $\text{AUTH}(\mathbb{N})$. 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:

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

  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 $\mathrm{Ex}(\mathrm{Unit})$.

- Notice that unlike other RA constructions, $\mathrm{Ex}(S)$ 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

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

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 $\texttt{()} \in \mathrm{Ex}(\mathrm{Unit})$, 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:

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

  Notice that even though only one element of $\text{Ex}(\text{Unit})$ 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` $l\{P * \texttt{()}\}$

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

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

  To release the lock, we need to show that P continues to hold, and that we have the key `()`.

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