iris-atomic

Zhen Zhang

December 22, 2016

1 Overview

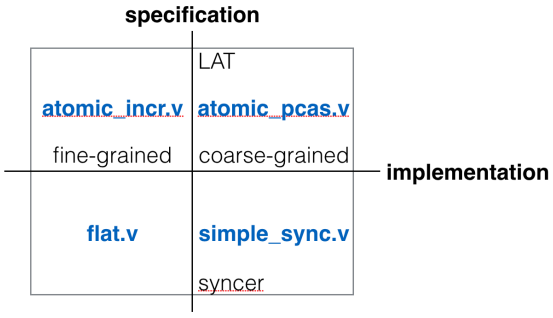

Figure 1: Two dimensions of atomicity verification in Iris

2 Generic syncer spec

$$\text{synced}(R, f', f) \triangleq \forall P, Q, x.\, \{R * P(x)\}\, f(x)\, \{v.\, R * Q(x, v)\} \rightarrow \{P(x)\}\, f'(x)\, \{v.\, Q(x, v)\}$$

$$\text{syncer}(R, s) \triangleq \forall f.\, \mathsf{wp}\, s(f)\, \{f'.\, \text{synced}(R, f', f)\}$$

$$\text{mkSyncer}(f) \triangleq \forall R.\, \{R\}\, f()\, \{s.\, \square \text{syncer}(R, s)\}$$

This **generic syncer spec** is inspired by CaReSL. In CaReSL, fine-grained syncer (e.g. flat combiner) is proven to be contextual refinement of coarse-grained `mkSync` (See section 4). However, there is no program refinement yet in Iris, so I made this generalization and prove that flat combiner satisfies this *directly*.

In fact, any sensible syncer is expected to satisfy this generic spec.

Besides generalization, I also split original spec into two parts, corresponding to two calls that happen at different times. More generally speaking, this problem is caused by a special case of curried function (e.g. $\lambda x_1.\, \lambda x_2.\, ...$): Is it possible to go from

$$\forall x_1, x_2.\, \{P(x_1, x_2)\}\, f(x_1, x_2)\, \{Q(x_1, x_2)\}$$

to

$$\forall x_1.\,\{\top\}\,f(x_1)\,\{f_1.\,\forall x_2.\,\{P(x_1,x_2)\}\,f_1(x_2)\,\{Q(x_1,x_2)\}\}$$

, assuming that $f \triangleq \lambda x_1, x_2.\, e$.

Finally, I'd like to point out that pre- and post-conditions P, Q are universally qualified per application, that is to say, P, Q are arbitrary depending on the context.

And similarly, f is universally qualified per synchronization, which means that syncer is only parameterized by the shared resource R, it can work with any operation that needs to access this R.

3 Logically atomic triple (LAT)

$$\langle g.\,\alpha(g)\rangle\, e\, \langle v.\,\beta(g,v)\rangle^{E_o}_{E_i} \triangleq$$
$$\forall P, Q.\ P \overset{Eo}{\Rightarrow}{}^{Ei}\ \exists g, \alpha(g) * (\alpha(g) \overset{Ei}{\Rightarrow}\!\!*^{Eo}\ P \wedge \forall v.\ \beta(g,v) \overset{Ei}{\Rightarrow}\!\!*^{Eo}\ Q(g,v)) \mathrel{-\!\!*}$$
$$\{P\}\, e\, \{v.\,\exists g.\, Q(g,v)\}$$

Note again, the P, Q are arbitrary depending on the context.

And also, note the separation product of two linear view shifts: First one represents **abandon** direction, while the second one represents **commit** direction. Using linear view shifts enables us to frame alongside the transition some non-persistent resource.

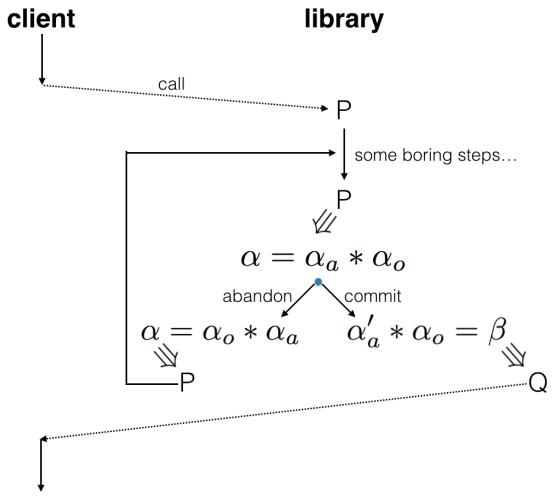

Figure 2: Sketch of how library function specified with LAT will be used

4 Coarse-grained syncer

First, we give the CaReSL's `mkSync` written in heap-lang (which is basically the same as in Coq, despite a few cosmetics).

```
mk_sync :=
    λ: <>,
```

```
let l := newlock() in
λ: f x,
   acquire l;
   let ret := f x in
   release l;
   ret.
```

Okay, now the question is: can we give a LAT spec for `mk_sync`? Actually, we can! and we can do this in a general way (i.e. for any syncer satisfying the generic syncer spec). We will briefly discuss about it in the next section.

5 Encoding generic syncer spec as a LAT-style spec

The first question is: **Why do we want to do so?**

First, they both talk about atomicity, so we will naturally want to know what is the possible relationship between them. And second, the generic syncer spec is not canonical – it doesn't tell explicitly what effects that operation has. When use the generic syncer spec as a client, you have to prove something about e every time. But LAT, on the contrary, gives you the exact specification of the operation itself, which means that you only need to prove the viewshifts (in some sense similar to weakening/strengthening rules). And most importantly, this exact specification, should look exactly like the sequential specification! Just consider the following illustrative comparison of a sequential/concurrent stack `push` spec:

$$\forall xs.\, \{stack(s, xs)\}\, \texttt{push}(x, s)\, \{\_.\, stack(s, x :: xs)\}$$
$$\langle xs.\, stack'(s, xs)\rangle\, \texttt{push'}(x, s)\, \langle \_.\, stack'(s, x :: xs)\rangle$$

The second question is: **Is it possible to do so?**

Intuitively, what syncer spec says is that the e's sequential effects are *compressed* to a single point. Consider the coarse-grained implementation of generic syncer spec, `mk_sync`, it guards the resource R by putting it inside a lock, thus every operation will exclusively own R for a certain period, which makes the whole operation look atomic, when observed from outside in terms of resource accessing.

The third question is: **How to do it?**

First, consider what LAT's client can provide to the library: abandoning and committing viewshifts. And these two actions must happen instantly. So, naturally, we will try to commit at the time of finishing the operation and give back the R to lock. But what about abandoning? What role will it have in the syncer spec?

Remember that there is a canonical pre-condition α in LAT, which should coincide with the sequential spec. But the sequential code, when executed between the lock/unlock, is still non-atomic. So after we exchange P for α, we must retain α for a long time, while we also have to close the LAT viewshifts in an instant. We can't close by abandoning, since we need to give back the α; and we can't close by committing, since we haven't started the action yet!

But wait, why we *must* give back α while losing it? What if we require that α is duplicable? Then problem solved! We just need to abandon in the beginning to get a duplicated α out, then committing with P when finishing off.

Now, with a sketch in mind, I will introduce you to the detailed construction: First, here is the specialized logically atomic triple to serve this purpose:

$$gtriple(\gamma, \alpha, f, x, \beta, E_i, E_o) \triangleq \left\langle g. \boxed{g^{1/2}}^{\gamma} * \square\, \alpha(g) \right\rangle f(x) \left\langle v. \exists g'. \boxed{g'^{1/2}}^{\gamma} * \beta(x, g, g', v) \right\rangle^{E_o}_{E_i}$$

Here, g represents **ghost state**. g can be of any proper type. $\boxed{g'^{1/2}}^{\gamma}$ is a snapshot of g, i.e. local knowledge about the physical configuration. (Why it is written in such way? Well, this is the implementation detail related to the monoid used to encode this).

Here, the persistent (thus duplicable) $\alpha(g)$ is some extra condition. And in the post condition, we say there exists an updated ghost state g', which satisfy β, a relation over input, previous state, current state, and output.

Let's take concurrent `push` again as an example. If s is a stack pointer,

$$R_s \triangleq \exists xs.\, stack(s, xs) * \boxed{xs^{1/2}}^{\gamma}$$

is the physical configuration plus global snapshot, then synced($stack(s, xs)$, `push'`, `push`(s)) should expand like this:

$$\begin{aligned} \forall P, Q, x.\, \{R_s * P(x)\}\, \mathsf{push}(s)(x)\, \{v.\, R_s * Q(x, v)\} \rightarrow \\ \{P(x)\}\, \mathsf{push'}(x)\, \{v.\, Q(x, v)\} \end{aligned}$$

And the LAT we can get from it looks like this:

$$\left\langle xs. \boxed{xs^{1/2}}^{\gamma} \right\rangle \mathsf{push'}(x) \left\langle \_. \exists xs'. \boxed{xs'^{1/2}}^{\gamma} * (xs' = x :: xs) \right\rangle$$

Here, note that in this case $\square\, \alpha(g)$ is selected as $\top$ (thus we can imagine that in most cases persistent restriction is not much of a problem); also, s is entirely hidden somewhere in some global place, i.e., using $\boxed{xs^{1/2}}^{\gamma}$, you can also atomically access $\exists s.\, stack(s, xs)$, even though such accessing is not specified as a LAT.

Now, let's consider the last question: **How to formalize it?**

```
sync(mk_syncer) :=
  λ: f_seq l,
     let s := mk_syncer() in
     s (f_seq l).
```

The code above is a helper function `sync`, which constructs the syncer, partially applies internal state value `l` to the sequential operation `f_seq`, and synchronizes the partially applied operation. There is an assumption that `l` should represent a valid state (either freshly constructed or whatever).

Now, if conditions $seqSpec(\mathtt{f\_seq}, \phi, \alpha, \beta, \top)$ (see 5) and $mkSyncer(\mathtt{mk\_syncer})$ are satisfied, then we have:

$$\forall g_0.\, \phi(l, g_0) \vdash \mathsf{wp}\, \mathsf{sync}(\mathtt{mk\_syncer}, \mathtt{f\_seq}, l)\, \{f.\, \exists \gamma. \boxed{g_0^{1/2}}^{\gamma} * \forall x.\, \square\, gtriple(\gamma, \alpha, f, x, \beta, E_i, \top)\}$$

Here is the definition of sequential specification pattern :

$$seqSpec(f,\phi,\alpha,\beta,E) \triangleq \forall l.\,\{\top\}\,f(l)\left\{f'.\;\begin{array}{l}\blacksquare\forall x,\Phi,g.\\ \quad \phi(l,g) * \square\,\alpha(g) *\\ \quad (\forall v,g'.\,\phi(l,g') \twoheadrightarrow \beta(x,g,g',v) \;{}^{E}\!\!\Rrightarrow\!\!\!\!\!\!\ast\;{}^{E}\;\Phi(v))\\ \quad \vdash \mathsf{wp}_E\, f'(x)\,\{\Phi\}\end{array}\right\}$$

In pre-condition, we have persistent $\alpha(g)$ and exclusive ownership of shared state; and when it returns, we get back the updated physical state g', as well as β.

Funny thing is again that I have to apply f to l first ... because of currying problem.

To be more concrete, here is the invariant that will be used globally: $(\exists g.\,\phi(l,g') * \boxed{g^{1/2}}^{\gamma} * Px$.

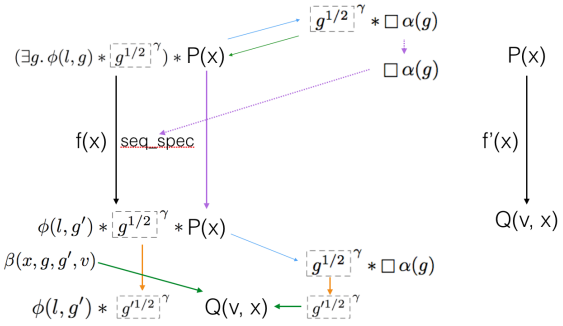

Figure 3: The sketch of proof: to prove triple for `f'(x)`, we prove another one for `f(x)`. The blue lines are opening viewshifts; the green lines are closing viewshifts; the orange lines are ghost update

6 Treiber's stack

```
push s x :=
  let hd := !s in
  let s' := ref SOME (x, hd) in
  if CAS s hd s'
    then ()
    else push s x.

pop s :=
  let hd := !s in
  match !hd with
  | SOME (x, hd') =>
    if: CAS s hd hd'
      then SOME x
      else pop s
  | NONE => NONE
  end.
```

```
iter hd f :=
  match !hd with
  | NONE => ()
  | SOME (x, hd') => f x ; iter hd' f
  end.
```

These `push` and `pop` are all standard lock-free implementations. We will give LAT spec for these two operations first:

6.1 Logically atomic spec (version 1)

$$\langle xs.\, stack(s, xs)\rangle\, push(s, x)\, \langle stack(s, x :: xs)\rangle^{\top}_{heapN}$$

$$\langle xs.\, stack(s, xs)\rangle\, pop(s) \left\langle v. \begin{array}{l} (\exists x, xs'.\, v = SOME(x) * stack(s, xs'))\vee \\ (v = NONE * xs = \emptyset * stack(s, \emptyset)) \end{array} \right\rangle^{\top}_{heapN}$$

6.2 Logically atomic spec (version 2)

$$\langle hd, xs.\, s \mapsto hd * list(hd, xs)\rangle\, push(s, x)\, \langle \exists hd'.\, s \mapsto hd' * hd' \mapsto SOME(x, hd) * list(hd, xs)\rangle^{\top}_{heapN}$$

$$\langle hd, xs.\, s \mapsto hd * list(hd, xs)\rangle\, pop(s) \left\langle v. \begin{array}{l} (\exists x, xs', hd'.\ \begin{array}{l} v = SOME(x) * hd \mapsto SOME(x, hd')* \\ s \mapsto hd' * list(hd', xs') \end{array})\vee \\ (v = NONE * xs = \emptyset * hd \mapsto NONE) \end{array} \right\rangle^{\top}_{heapN}$$

6.3 Why such dichotomy?

Apparently, the version 1 is more preferable as a general spec, since the implementation might not necessarily be a linked-list.

But if we want to iterate over it using `iter` using a per-item style spec, than we might expose enough implementation details to tie per-item resource to physical location. And as a result, to maintain such property, we must also make `push`'s LAT spec exposing enough details as well (And it doesn't even make sense to use `pop` in this case). Below is the discussion about per-item spec.

7 Per-item spec

$$fSpec(R, f, Rf, x) \triangleq \left\{\boxed{R}^{\iota} * Rf\right\} f(x) \{v.\, v = () * Rf\}$$

Note that R is *not* parametric, while this spec is parametric with x. So, the R here is expected to be a fully applied invariant. During the proof, we will actually only need to consider one such item, then inductively prove the iteration's property. So such a bizarre design can work. And most importantly, any item-associated thing, like ghost tokens, can be exposed.

Now, let's observe the `push`'s spec (in a per-item setting):

$$pushSpec(s, x) \triangleq \left\{R(x) * \boxed{\exists xs.\, stack'(xs, s)}^{\iota}\right\} push(s, x) \left\{v.\, v = () * \boxed{R(x)}^{\iota}\right\}$$

Okay, back to the per-item spec, why do you need to make sure that the stack is growth-only, and governed by a global invariant?

Answer: because we need to iterate thought it non-atomically. I mean, the process of iteration is not atomic. The single operation `f` still needs to atomically access any resource related `x`.

8 Flat combiner

Here is my implementation:

```
doOp :=
  λ: p,
     match !p with
     | InjL (f, x) => p <- InjR (f x)
     | InjR _ => ()
     end.

try_srv :=
  λ: lk s,
    if try_acquire lk
      then let hd := !s in
           iter hd doOp;
           release lk
      else ().

loop p s lk :=
    match !p with
    | InjL _ =>
        try_srv lk s;
        loop p s lk
    | InjR r => r
    end.

install :=
  λ: f x s,
     let p := ref (InjL (f, x)) in
     push s p;
     p.

mk_flat :=
  λ: <>,
   let lk := newlock() in
   let s := new_stack() in
   λ: f x,
      let p := install f x s in
      let r := loop p s lk in
      r.
```

Compared to CaReSL, here are several notable differences:

- I break them down into five procedures, instead of nesting them inside the constructor. They all have their own spec, and it is easier to reason about.
- CaReSL's flat constructor takes `f`; mine doesn't, which is more flexible.
- CaReSL's install utilizes TLS (thread-local storage) to avoid severe memory leakage. In my implementation, a new slot is created and pushed every time the operation is called. It is terrible in practice, but nevertheless follows the same spec.

Also, another problem that exists both in my, CaReSL, and FCSL's example code, is that we never recycle the slots.

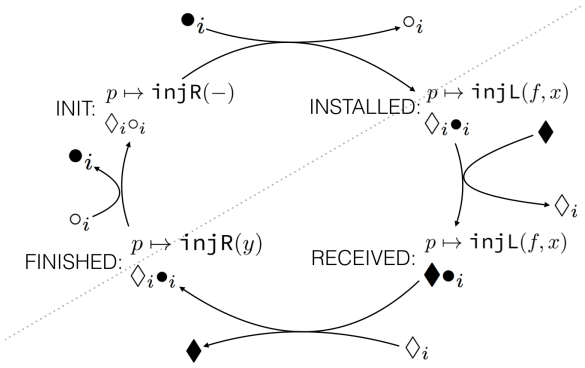

Figure 4: Helping protocol illustration. Note that this graph is heavily simplified to reflect how I encode protocol with bare exclusive monoids.

The black diamond is a token owned by server lock; The other three kinds of token are all tagged with i, which is similar to thread id in CaReSL's reasoning, but in my case, it is more flexible by using a simple existential qualification.

Here is the per-item (i.e. request slot) invariant, which have four branches.

$$
\begin{array}{ll}
\exists y. & p \overset{1/2}{\longmapsto} \mathtt{injR}(-) * \Diamond_i * \circ_i \\
\vee \exists f, x, P, Q. & p \overset{1/2}{\longmapsto} \mathtt{injL}(f, x) * \boxed{x^{1/2}}^{\gamma} * P(x) * (\{R * P(x)\}\, f(x)\, \{v.\, R * Q(x, v)\}) * \kappa \Rrightarrow Q(x) * \Diamond_i * \bullet_i \\
\vee \exists x. & p \overset{1/2}{\longmapsto} \mathtt{injL}(-, x) * \boxed{x^{1/4}}^{\gamma} * \blacklozenge * \bullet_i \\
\vee \exists x, y. & p \overset{1/2}{\longmapsto} \mathtt{injR}(y) * \boxed{x^{1/2}}^{\gamma} * \kappa \Rrightarrow Q(x) * Q(x, y) * \Diamond_i * \bullet_i
\end{array}
$$

Note how f, x, P, Q are all hidden under existential qualification, and $Q(x)$ is saved under some constant name κ. Note how only post-condition matters to waiting client.