FORMALIZING CONCURRENT STACKS WITH HELPING: A CASE STUDY IN IRIS

DANIEL GRATZER, MATHIAS HØIER, ALEŠ BIZJAK, AND LARS BIRKEDAL

Abstract. Iris is an expressive higher-order separation logic designed for the verification of concurrent, imperative programs. As a demonstration of this, we present a formalization of a common concurrent data-structure: a stack. The implementation is lock-free and uses helping to avoid contention. All of the examples are formalized in Coq and demonstrate how Iris can be used to give expressive, higher-order specifications of advanced concurrent data-structures.

1. Introduction

Iris is a general, base logic suitable for constructing a variety of program logics. It has been used extensively to verify properties of various programs, in particular those making use of fine-grained concurrency. In this case study we will verify a larger example of such a concurrent program which uses atomic operations to implement a lock-free data-structure.

Despite how it is normally used, the Iris logic itself is designed quite independently of any particular programming language or indeed the notion of a program logic at all. It is a standard higher-order logic supplemented with several modalities. An introduction to it may be found in Jung et al. [2017] or Jung et al. [2015]. In this paper we will assume familiarity with Iris as well as one of the main languages it has been used to study: a concurrent ML supplemented with general, mutable reference and a CAS operation. We will focus on showing how to use the logic provided by Iris to examine a real, substantive concurrent data-structure: a concurrent stack with helping. We hope that this case study will give a flavor for how verification of such data-structures normally proceeds in Iris. In order to do this we will give several different specifications for the stack. Each specification will be more precise and correspondingly complex than the previous one. Inevitably, the specifications will force us to choose between the modularity of the proof and the precision with which our specification describes the data-structure. The first specification, which ignores the stack ordering of the data-structure and merely treats it as a bag, allows for a simple and direct verification that follows the abstractions set up in the code itself. The final specification which captures far more of the behavior of the stack forces us to ignore such abstractions.

To begin with, we will start by reviewing the code and informally arguing towards its correctness. We will then specify it as a bag and provide a fully worked out proof that the code satisfies the specification. We will then prove a more precise specification making use of atomic updates to an abstract stack for our data structure with and without helping. Finally, we will conclude with the same precise specification applied to the full data-structure supplemented with helping.

Carnegie Mellon University
Aarhus University
Date: March 13, 2018.

2. A Concurrent Stack with Helping

The abstract data type that we're implementing is that of a stack. Therefore, it will have two operations, `push` and `pop`. The main complication of our data structure is that `push` and `pop` must be thread-safe.

In order to handle this in the implementation, we will make use of the fact that we have an atomic compare-and-swap operation on reference cells in our language. By storing our stack in such a cell, we can retrieve it, modify it as necessary, and attempt to atomically replace the old stack with our new stack. As is usual with fine-grained concurrency, it is entirely possible that halfway through our operation another thread might modify the stack. In this case, the attempt to replace the old stack with the new stack using a CAS instruction will fail. The typical way to handle this is to loop and attempt the operation from the beginning, though in high contention situations this can be a major delay and cause long running operations to live-lock.

In order to partially mitigate this, it would be ideal if some of our concurrent operations could avoid dealing with the main cell holding the stack entirely. For example, if one thread is attempting a push at the same time another is attempting a pop, they will fight each other for ownership of the cell even though it would be perfectly valid for one thread to simply hand the other the value. In a high contention situation with many threads pushing and popping at once, this can be quite a common problem. We handle this by introducing a *side-channel* for threads to communicate along. That is, before a thread attempts to work with the main stack it will, for instance, check whether or not someone is offering a value along the side-channel that it could just take instead. Similarly a thread which is pushing a value onto the stack will offer its value on the side-channel temporarily in an attempt to avoid having to compete for the main stack. This scheme reduces contention on the main atomic cell and thus improves performance.

2.1. **Mailboxes for Offers.**

In order to do this, before designing a stack we first implement a small API for side-channels. A side-channel has the following operations

(1) An offer can be *created* with an initial value.
(2) An offer can be accepted, marking the offer as taken and returning the underlying value.
(3) Once created, an offer can be revoked which will prevent anyone from accepting the offer and return the underlying value to the thread.

Of course, all of these operations have to be thread-safe! That is, it must be safe for an offer to be attempted to be accepted by multiple threads at once, an offer needs to be able to be revoked while it's being accepted, and so on. We choose to represent an offer as a tuple of the actual value the offer contains and a reference to an int. The underlying integer may take one of 3 values, either 0, 1 or 2. Therefore, an offer of the form (v, ℓ) with $\ell \mapsto 0$ is the initial state of an offer, no one has attempted to take it nor has it been revoked. Someone may attempt to take the offer in which case they will use a CAS to switch ℓ from 0 to 1, leading to the accepted state of an offer which is (v, ℓ) so that $\ell \mapsto 1$. Revoking is almost identical but instead of switching from 0 to 1 instead we switch to 2. Since both revoking and accepting an offer demand the offer to be in the initial state it is impossible anything other than exactly one accept or one revoke to succeed. The actual code for this is in Figure 1 and the transition graph sketched above is illustrated in Figure 2. The pattern of offering

```
mk_offer = fun v → (v, ref 0)
revoke_offer =
  fun v →
    if cas (snd v) 0 2
    then Some (fst v)
    else None
accept_offer =
  fun v →
    if cas (snd v) 0 1
    then Some (fst v)
    else None
```

FIGURE 1. The implementation of offers used to construct side-channels.

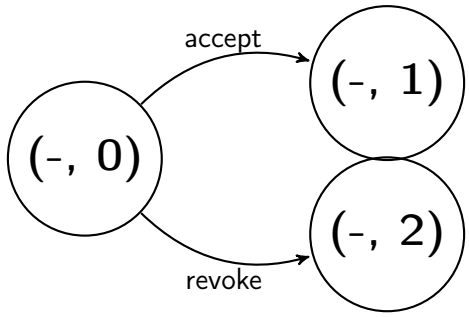

FIGURE 2. The states an offer may be in.

something, immediately revoking it, and returning the value if the revoke was successful is sufficiently common that we can encapsulate it in an abstraction called a *mailbox*. The idea is that a mailbox is built around an underlying cell containing an offer and that it provides two functions which, respectively, briefly put a new offer out and check for such an offer. The code for this may be found in Figure 3. One small technical detail is that we have designed this as a constructor which returns two closures which manipulate the same reference cell. This simplifies the process of using a mailbox for stacks where we only have one mailbox at a time. It is not, however, fundamentally different than an implementation more in the style of the offers above.

2.2. **The Implementation of the Stack.** With an implementation of offers, it is easy to code up the concurrent stack. The idea for implementing a side-channel is to have a designated cell for threads to put pending offers in. This way, when a thread comes along to push a value onto the stack it will first create an offer using the above API and put it into the given cell. It will then immediately revoke it to see if another thread has accepted the offer in the meantime and if none has, it will proceed with the pushing algorithm. The dual process suffices for a thread seeking to pop. The code for the stack is in Figure 4. Notice that this too is written in a similar style to that of mailboxes, a make function which returns two closures for the operations rather than having them separately accept a stack as an argument.

```
mailbox = fun () →
  let r = ref None in
  (rec put v →
     let off = mk_offer v in
     r := Some off;
     revoke_offer off,
   rec get n →
     let offopt = !r in
     match offopt with
       None → None
     | Some x → accept_offer x
     end)
```

FIGURE 3. The implementation of mailboxes which provide a convenient wrapper over offers.

3. A First Formalization: A Bag Specification

The first specification of the concurrent stack really only specifies the stack's behavior as a *bag*. Nowhere in the specification is the order of insertion reflected. In a concurrent setting this is less damning than it might appear because from the perspective of a single thread it is indeed the case that there is little connection between the order in which things are inserted and when they will be removed. This is a direct result of the fact that a thread must be agnostic to the interference and actions of other threads operating on the same stack concurrently.

With this in mind, the specification for the stack will be parameterized by some arbitrary predicate on values, $P : \mathit{Val} \to \mathit{iProp}$. Every element of the stack will satisfy P and thus our specifications are roughly

$$\begin{aligned} P(v) \mathrel{-\!\!*} & \mathsf{wp}_{\mathcal{E}}\, \mathsf{push}(v)\, \{\mathsf{True}\} \\ & \mathsf{wp}_{\mathcal{E}}\, \mathsf{pop}() \quad \{v.v = \mathsf{None} \lor \exists v'.\, v = \mathsf{Some}(v') * P(v')\} \end{aligned}$$

Of course the actual specifications must specify `stack`, a function which returns a tuple a push and pop. This necessitates the use of a higher order specification, a weakest preconditions whose post condition contains other weakest preconditions. This is possible because Iris weakest preconditions are not encoded as a separate sort of logical proposition but rather as ordinary *iProp*s. Furthermore, to make using the specification easier, a common trick with weakest preconditions is employed. Instead of directly stating something along the lines of $\mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}$, instead we introduce a "cut", $\forall \Psi.\, (\forall v.\, \Phi(v) \mathrel{-\!\!*} \Psi(v)) \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}}\, e\, \{\Psi\}$. This makes chaining together multiple weakest preconditions considerably simpler and avoids gratuitous uses of the rule of consequence. With all of this said, the first specification for concurrent

```
stack = fun () →
  let mailbox = mailbox () in
  let put = fst mailbox in
  let get = snd mailbox in
  let r = ref None in
  (rec pop n →
     match get () with
       None →
       (match !r with
          None → None
        | Some hd =>
          if cas r (Some hd) (snd hd)
          then Some (fst hd)
          else pop n
        end)
     | Some x → Some x
     end,
   rec push n →
     match put n with
       None → ()
     | Some n →
       let r' = !r in
       let r'' = Some (n, r') in
       if cas r r' r''
       then ()
       else push n
     end)
```

FIGURE 4. The implementation of the concurrent stack.

stacks is[1]

$$
\begin{aligned}
&\forall \Phi.\\
&(\forall f_1 f_2.\ \Box\, \mathsf{wp}\ f_1()\,\{v. v = \mathsf{None} \lor \exists v'.\, v = \mathsf{Some}(v') * P(v')\}\\
&\qquad -\!\!*\forall v.\ \Box(P(v) -\!\!* \mathsf{wp}\ f_2(v)\,\{\mathsf{True}\})\\
&\qquad -\!\!*\Phi(f_1, f_2))\\
&\quad -\!\!* \mathsf{wp}\ \mathsf{stack}()\,\{\Phi\}
\end{aligned}
$$

[1]The $\Box$-modalities ensures that the specifications for f_1 and f_2 are persistent. Without these the specification would be quite weak, as one would only be allowed to call push and pop once each.

Rather than directly verifying this specification, the proof depends on several helpful lemmas verifying the behavior of offers and mailboxes. By proving these simple sublemmas the verification of concurrent stacks can respect the abstraction boundaries constructed by isolating mailboxes as we have done.

3.1. **Verifying Offers.** The heart of verifying offers is accurately encoding the transition system described in the previous section. Roughly, an offer can transition from initial to accepted or from initial to revoked but no other transitions are to be allowed. Encoding this requires a simple but interesting application of ghost state. It is possible to encode an arbitrary state transition system in Iris but in this case a more specialized approach is simpler.

Specifically, offers will be governed by a proposition stages which encodes what state of the three an offer is in. Ghost state is needed to ensure that certain transitions are only possible for threads with *ownership* of the offer. To do this, the exclusive monoid on unit will act as a token giving the owner the right to transition to the original state to the revoked state. The main thread will have access to $\boxed{\mathsf{ex}(())}^{\gamma}$ for some γ and it will be necessary to use this resource to transition from the initial state to the revoked state. The proposition encoding the transition system is

$$\mathsf{stages}_{\gamma}(v,\ell) \triangleq (P(v) * \ell \mapsto 0) \vee \ell \mapsto 1 \vee (\ell \mapsto 2 * \boxed{\mathsf{ex}(())}^{\gamma})$$

Having defined this, the proposition is_offer is now within reach.

$$\mathsf{is\_offer}_{\gamma}(v) \triangleq \exists v', \ell.\, v = (v', \ell) * \exists \iota.\, \boxed{\mathsf{stages}_{\gamma}(v', \ell)}^{\iota}$$

Notice that is_offer is clearly persistent, reflecting the fact that it ought to be shared between multiple threads. This implies that knowing $\mathsf{is\_offer}_{\gamma}(v)$ does not assert ownership of any kind, rather, it asserts that for an atomic step of computation the owner may assume that the offer is in one of the three states. This sharing provides the motivation for using invariants to capture stages. Without wrapping it in an invariant it would not be possible to share it between multiple threads. Notice that both of these propositions are parameterized by a ghost name, γ. Each γ should uniquely correspond to an offer and represents the ownership the creator of an offer has over it, namely the right to revoke it. This is expressed in the specification of `mk_offer`.

$$\forall v.\, P(v) \twoheadrightarrow \mathsf{wp}\ \mathsf{mk\_offer}(v) \left\{ v.\ \exists \gamma.\, \boxed{\mathsf{ex}(())}^{\gamma} * \mathsf{is\_offer}_{\gamma}(v) \right\}$$

This reads as that calling `mk_offer` will allocate an offer *as well as* returning $\boxed{\mathsf{ex}(())}^{\gamma}$ which represents the right to revoke an offer. This can be seen in the specification for `revoke_offer`.

$$\forall \gamma, v.\, \mathsf{is\_offer}_{\gamma}(v) * \boxed{\mathsf{ex}(())}^{\gamma} \twoheadrightarrow \mathsf{wp}\ \mathsf{revoke\_offer}(v)\, \{v.\ v = \mathsf{None} \vee \exists v'.\, v = \mathsf{Some}(v) * P(v')\}$$

The specification for `accept_offer` is remarkably similar except that it does not require ownership of $\boxed{\mathsf{ex}(())}^{\gamma}$. This is because multiple threads may call `accept_offer` even though it will only successfully return once.

$$\forall \gamma, v.\, \mathsf{is\_offer}_{\gamma}(v) \twoheadrightarrow \mathsf{wp}\ \mathsf{accept\_offer}(v)\, \{v.\ v = \mathsf{None} \vee \exists v'.\, v = \mathsf{Some}(v) * P(v')\}$$

As an illustrative example, we will go through the the derivation of the specification for mk_offer and leave the derivations of the other two specifications as an exercise to the reader.

Theorem 3.1. $\forall v.\, P(v) \mathrel{-\!\!*} \mathit{wp}\ \mathit{mk\_offer}(v) \left\{ v.\ \exists \gamma.\, \boxed{\mathsf{ex}(())}^{\gamma} * \mathit{is\_offer}_{\gamma}(v) \right\}$ *holds.*

Proof. Let us assume that we have some $v \in \mathit{Val}$ and further that $P(v)$ holds. We wish to show that $\mathsf{wp}\ \mathsf{mk\_offer}(v) \left\{ v.\ \exists \gamma.\, \boxed{\mathsf{ex}(())}^{\gamma} * \mathsf{is\_offer}_{\gamma}(v) \right\}$ holds. By applying the β rule for functions, we must show that

$$\mathsf{wp}\ (v, \mathsf{ref}(0)) \left\{ v.\ \exists \gamma.\, \boxed{\mathsf{ex}(())}^{\gamma} * \mathsf{is\_offer}_{\gamma}(v) \right\}$$

We then apply the rule for allocation, so we may assume that we have some location ℓ and that $\ell \mapsto 0$. We now need to show that $\mathsf{wp}\ (v, \ell) \left\{ v.\ \exists \gamma.\, \boxed{\mathsf{ex}(())}^{\gamma} * \mathsf{is\_offer}_{\gamma}(v) \right\}$. Now we merely need to show that

$$(v, \ell) \in \mathit{Val} \wedge \Rrightarrow_{\mathcal{E}} \exists \gamma.\, \boxed{\mathsf{ex}(())}^{\gamma} * \mathsf{is\_offer}_{\gamma}((v, \ell))$$

The left-hand side of this conjunction is trivial. For the right-hand side, we note that since $\mathcal{V}(\mathsf{ex}(()))$ clearly holds we have that $\Rrightarrow_{\mathcal{E}} \exists \gamma.\, \boxed{\mathsf{ex}(())}^{\gamma}$ holds. Therefore, it suffices to show that $\Rrightarrow_{\mathcal{E}} \mathsf{is\_offer}_{\gamma}((v, \ell))$ holds for some γ. For this, we first note that $\mathsf{is\_offer}_{\gamma}((v, \ell))$ is of course equal to $\exists v', \ell'.\, (v, \ell) = (v', \ell') * \exists \iota.\, \boxed{\mathsf{stages}_{\gamma}(v', \ell')}^{\iota}$ So we start by introducing the existential quantifier with v and ℓ giving us the goal

$$\Rrightarrow_{\mathcal{E}} (v, \ell) = (v, \ell) * \exists \iota.\, \boxed{\mathsf{stages}_{\gamma}(v, \ell)}^{\iota}$$

The equality is obviously true so we merely need to show that $\Rrightarrow_{\mathcal{E}} \exists \iota.\, \boxed{\mathsf{stages}_{\gamma}(v, \ell)}^{\iota}$. For this, we will use the invariant allocation rule, so we need to show that $\triangleright\, \mathsf{stages}_{\gamma}(v, \ell)$ holds and we're done. For this, we prove $\mathsf{stages}_{\gamma}(v, \ell)$ for which it suffices to show $P(v) * \ell \mapsto 0$ which we have in our assumptions. □

3.2. **Verifying Mailboxes.** Having verified that offers work as intended, the next step is to verify that the mailbox abstraction built on top of them also satisfies the intended specification. In order to properly specify mailboxes, it is necessary to use a similar trick to that of the specification of stacks. That is, a specification that involves higher order weakest preconditions and bakes in a cut.

(1)
$$\begin{aligned} &\forall \Phi. \\ &\quad (\forall f_1 f_2.\, (\forall v.\, \Box(P(v) \mathrel{-\!\!*} \mathsf{wp}\ f_1(v)\, \{v.v = \mathsf{None} \vee \exists v'.\, v = \mathsf{Some}(v') * P(v')\})) \\ &\qquad \mathrel{-\!\!*} \Box\, \mathsf{wp}\ f_2()\, \{v.v = \mathsf{None} \vee \exists v'.\, v = \mathsf{Some}(v') * P(v')\} \\ &\qquad \mathrel{-\!\!*} \Phi(f_1, f_2)) \\ &\quad \mathrel{-\!\!*} \mathsf{wp}\ \mathsf{mailbox}()\, \{\Phi\} \end{aligned}$$

In a small victory for compositional verification, the proof of this specification is made with no reference to the underlying implementation of offers, only to the specification previously proven. Throughout the proof an invariant is maintained governing the shared mutable cell that contains potential offers. This invariant enforces that when this cell is full, it contains an offer. It looks like this

$$\mathsf{is\_mailbox}(v) \triangleq \exists \ell.\, v = \ell * \ell \mapsto \mathsf{None} \vee \exists v' \gamma.\, l \mapsto \mathsf{Some}(v') * \mathsf{is\_offer}_{\gamma}(v')$$

This captures the informal notion described above.

Theorem 3.2. *Proposition (1) holds.*

Proof. For this, we start by applying the β-rule. This means that in addition to our assumption that

$$\begin{aligned}&\forall f_1 f_2.\, (\forall v.\, \square(P(v) \mathrel{-\!\!*} \mathsf{wp}\, f_1(v)\, \{v.v = \mathsf{None} \vee \exists v'.\, v = \mathsf{Some}(v') * P(v')\})) \\ &\quad \mathrel{-\!\!*} \square\, \mathsf{wp}\, f_2()\, \{v.v = \mathsf{None} \vee \exists v'.\, v = \mathsf{Some}(v') * P(v')\} \\ &\quad \mathrel{-\!\!*} \Phi(f_1, f_2)\end{aligned}$$

our goal is

$$\mathsf{wp}\, \texttt{let r = ref None in (..., ...)}\, \{\Phi\}$$

We then apply the rule for allocation, so we suppose that we have some ℓ such that we also have $\ell \mapsto \mathsf{None}$. Our goal after applying another β rule is then of the form

$$\mathsf{wp}\, (..., ...)\, \{\Phi\}$$

We now need to show that $\Rrightarrow_\top \Phi(..., ...)$ holds so we take this time to allocate the mailbox invariant as discussed above. We will allocate $\boxed{\mathsf{is\_mailbox}(\ell)}^\iota$, for some ι, so we must prove that $\triangleright\, \mathsf{is\_mailbox}(\ell)$ holds. We first instantiate the existential quantifier with ℓ leaving us with the goal

$$\ell = \ell * \ell \mapsto \mathsf{None} \vee \exists v', \gamma.\, l \mapsto \mathsf{Some}(v') * \mathsf{is\_offer}_\gamma(v')$$

Obviously the equality holds and the left side of the disjunct holds by our assumption that $\ell \mapsto \mathsf{None}$ so we're done. We now apply our original hypothesis leaving us to prove

$$\begin{aligned}\forall v.\, \square(P(v) \mathrel{-\!\!*} &\mathsf{wp}\, f_1(v)\, \{v.v = \mathsf{None} \vee \exists v'.\, v = \mathsf{Some}(v') * P(v')\}) \\ &\square\, \mathsf{wp}\, f_2()\quad \{v.v = \mathsf{None} \vee \exists v'.\, v = \mathsf{Some}(v') * P(v')\}\end{aligned}$$

where we have defined f_1 and f_2 as

```
f1 = rec put v →
 let off = mk_offer v in
 r := Some off;
 revoke_offer off,
```

```
f2 = rec get n →
 let offopt = !r in
 match offopt with
   None → None
 | Some x →
     accept_offer x
 end
```

We shall verify the specification for f_1 and leave the specification of f_2 as an exercise. To eliminate the $\square$-modality, we have to throw away all non-persistent resources. As our context is just a single invariant, hence persistent, we can proceed without any losses. Let us now assume that we have $P(v)$ for some v. We then apply the specification for `mk_offer` to conclude that it suffices to show

$$\begin{aligned}&\mathsf{is\_offer}_\gamma(\mathsf{off}) * \boxed{\mathsf{ex}(())}^\gamma \mathrel{-\!\!*} \\ &\qquad \mathsf{wp}\, (\texttt{r := Some off; revoke\_offer off})\, \{v.v = \mathsf{None} \vee \exists v'.\, v = \mathsf{Some}(v') * P(v')\}\end{aligned}$$

So let us assume that we know $\mathsf{is\_offer}_\gamma(\mathsf{off})$ as well as $\boxed{\mathsf{ex}(())}^\gamma$. We then open our invariant for the single atomic reduction of `r := Some off`. We must then show the following

$$\triangleright\,\mathsf{is\_mailbox}(r) \mathrel{-\!\!*} \mathsf{wp}\,(\texttt{r := Some off})\left\{ v.\ \begin{array}{l} \mathsf{is\_mailbox}(r) * \\ \mathsf{wp}\,(\texttt{v; revoke\_offer off})\left\{ v.\ \begin{array}{l} v = \mathsf{None} \\ \vee\ \exists v'.\, v = \mathsf{Some}(v') * P(v') \end{array} \right\} \end{array} \right\}$$

Notice that we have only $\triangleright\,\mathsf{is\_mailbox}(r)$ because opening an invariant gives only $\triangleright$ of the stored proposition. This will suffice in our case because the rule for loading a location does not require $\ell \mapsto v$, only $\triangleright\,\ell \mapsto v$ because $\ell \mapsto v$ is a timeless proposition. Weakest preconditions are designed in such a way that it is always possible to remove $\triangleright$s from timeless propositions.

Let us then assume that we have $\triangleright\,\mathsf{is\_mailbox}(r)$. We know then that there is a location ℓ' so that $r = \ell'$ and $\triangleright\,\ell \mapsto \mathsf{None} \vee \exists v'\gamma.\, \triangleright(l \mapsto \mathsf{Some}(v') * \mathsf{is\_offer}_\gamma(v'))$. We then case analyze this disjunction.

In the first case, we have $\ell' \mapsto \mathsf{None}$ so we can apply the rule for stores leaving us with the goal

$$\ell' \mapsto \mathsf{Some}(\mathsf{off}) \mathrel{-\!\!*} \mathsf{is\_mailbox}(\ell') * \mathsf{wp}\,(\texttt{v; revoke\_offer off})\,\{v.v = \mathsf{None} \vee \exists v'.\, v = \mathsf{Some}(v') * P(v')\}$$

First, we prove $\mathsf{is\_mailbox}(\ell')$ with our assumptions that $\ell' \mapsto \mathsf{Some}(\mathsf{off})$ and $\mathsf{is\_offer}_\gamma(\mathsf{off})$. This is quite straightforward. We prove this existential for ℓ'. For this we must show that

$$\ell = \ell' * \ell \mapsto \mathsf{None} \vee \exists v'\gamma.\, l \mapsto \mathsf{Some}(v') * \mathsf{is\_offer}_\gamma(v')$$

but the right disjunct is precisely the assumptions we have. We then must show the rest of the goal

$$\mathsf{wp}\,(\texttt{(); revoke\_offer off})\,\{v.v = \mathsf{None} \vee \exists v'.\, v = \mathsf{Some}(v') * P(v')\}$$

For this we apply the β and notice that

$$\mathsf{wp}\,\texttt{revoke\_offer(off)}\,\{v.v = \mathsf{None} \vee \exists v'.\, v = \mathsf{Some}(v') * P(v')\}$$

is precisely the specification we proved earlier for `revoke_offer` and we conveniently have $\boxed{\mathsf{ex}(())}^\gamma$ and $\mathsf{is\_offer}(\mathsf{off})$ (remember that it's persistent) so we're done.

The reasoning for the other disjunct is identical so we elide it here. □

3.3. **Verifying Stacks.** We now turn to the verification of stacks themselves. The specification for these has already been discussed:

$$\begin{array}{l} \forall \Phi. \\ \quad (\forall f_1 f_2.\, \square\,\mathsf{wp}\, f_1()\,\{v.v = \mathsf{None} \vee \exists v'.\, v = \mathsf{Some}(v) * P(v)\} \\ \qquad \mathrel{-\!\!*} \forall v.\, \square(P(v) \mathrel{-\!\!*} \mathsf{wp}\, f_2(v)\,\{\mathsf{True}\}) \\ \qquad \mathrel{-\!\!*} \Phi(f_1, f_2)) \\ \quad \mathrel{-\!\!*} \mathsf{wp}\,\mathsf{stack}()\,\{\Phi\} \end{array} \tag{2}$$

Having verified mailboxes already only a small amount of additional preparation is needed before actually verifying this proposition. Specifically, an invariant representing that a memory cell contains a stack is needed. This is necessary because this cell will be shared between

multiple threads concurrently reading and writing to it and without an invariant there is no way to reason about this. The predicate $\mathsf{is\_stack}(v)$ used to form the invariant is defined as follows by guarded recursion

$$\mathsf{is\_stack}(v) \triangleq \mu\, R.\, v = \mathsf{None} \vee \exists h, t.\, v = \mathsf{Some}(h, t) * P(h) * \triangleright\, R(t)$$

Having defined this, it is straightforward to define an invariant enforcing that a location points to a stack.

$$\mathsf{stack\_inv}(v) \triangleq \exists \ell, v'.\, v = \ell * \ell \mapsto v' * \mathsf{is\_stack}(v')$$

We turn now to verifying the proposition.

Theorem 3.3. *Proposition (2) holds.*

Proof. For this, we assume first that we have

$$\begin{aligned}\forall f_1 f_2.\ &\square\, \mathsf{wp}\ f_1()\ \{v. v = \mathsf{None} \vee \exists v'.\, v = \mathsf{Some}(v) * P(v)\} \\ &-\!\!* \square(\forall v.\, P(v) -\!\!* \mathsf{wp}\ f_2(v)\ \{\mathsf{True}\}) \\ &-\!\!* \Phi(f_1, f_2))\end{aligned}$$

Now we apply the β rule to conclude that we must show that

$$\mathsf{wp}\,(\texttt{let mailbox = mailbox () in E})\,\{\Phi\}$$

We can then apply the bind rule to see that it suffices to prove instead

$$\mathsf{wp}\,\texttt{mailbox}()\,\{v.\mathsf{wp}\,\texttt{let mailbox = v in E}\,\{\Phi\}\}$$

This is in the form that we can apply our specification for `mailbox`. Our goal is now to show that

$$\begin{aligned}\forall f_1 f_2.\, \forall v.\ &\square(P(v) -\!\!* \mathsf{wp}\ f_1(v)\ \{v. v = \mathsf{None} \vee \exists v'.\, v = \mathsf{Some}(v') * P(v')\}) \\ &-\!\!* \square\, \mathsf{wp}\ f_2()\ \{v. v = \mathsf{None} \vee \exists v'.\, v = \mathsf{Some}(v') * P(v')\} \\ &-\!\!* \mathsf{wp}\,\texttt{let mailbox = (f1, f2) in E}\,\{\Phi\}\end{aligned}$$

This is the advantage provided by specifying our lemmas with a cut built in. The mailbox lemma is immediately applicable without further manipulation. Let us assume that we have such an f_1 and f_2 and furthermore that the above two specifications holds for f_1 and f_2. We now attempt to prove that $\mathsf{wp}\,\texttt{let mailbox = (f1, f2) in E}\,\{\Phi\}$. We next apply the β rule for `let` to transform our goal into

$$\mathsf{wp}\,\texttt{let get = f1 in let put = f2 in let r = ref None in (P1, P2)}\,\{\Phi\}$$

Now again we can apply our β rules to conclude that we need to show that the following holds

$$\mathsf{wp}\,\texttt{let r = ref None in ([f1/get]P1, [f2/put]P2)}\,\{\Phi\}$$

We now apply the allocation rule, so suppose that we have some ℓ, we must then show that if $\ell \mapsto \mathsf{None}$ then $\mathsf{wp}\,\texttt{([f1/get]P1, [f2/put]P2)}\,\{\Phi\}$ holds. Before attempting to prove this though, we allocate then invariant $\mathsf{stack\_inv}(\ell)$. This is easy to do because allocating this invariant requires showing that

$$\triangleright\, \exists \ell', v'.\, \ell = \ell' * \ell' \mapsto v' * \mathsf{is\_stack}(v')$$

Let us prove this by proving that $\exists \ell' v'.\, \ell = \ell' * \ell' \mapsto v' * \mathsf{is\_stack}(v')$ directly and instantiating the existential with ℓ and None. Our remaining obligation is just to show that

$$\mu\, R.\, v = \mathsf{None} \vee \exists h, t.\, v = \mathsf{Some}(h, t) * P(h) * \triangleright R(t)$$

But the left disjunct is just reflexivity. We may then assume $\boxed{\mathsf{is\_stack}(r)}^{\iota}$. Now our goal is simply of the form

$$\mathsf{wp}\ (\texttt{[f1/get]P1, [f2/put]P2})\ \{\Phi\}$$

We, however, have an assumption that

$$\begin{aligned}\forall f_1 f_2.\ \square\, \mathsf{wp}\ f_1()\ \{v.v = \mathsf{None} \vee \exists v'.\, v = \mathsf{Some}(v') * P(v')\}\\ -\!\!*\forall v.\ \square\, P(v) -\!\!*\ \mathsf{wp}\ f_2(v)\ \{\mathsf{True}\}\\ -\!\!*\Phi(f_1, f_2))\end{aligned}$$

Now we apply this to our current goal leaving us to show that

$$\begin{aligned}&\square\mathsf{wp}\ \texttt{[f1/get]P1()}\ \{v.v = \mathsf{None} \vee \exists v'.\, v = \mathsf{Some}(v') * P(v')\}\\ \forall v.\ \square(P(v) -\!\!*&\mathsf{wp}\ \texttt{[f1/get]P2(v)}\ \{\mathsf{True}\})\end{aligned}$$

We will prove both of these separately now.

(1) $\square\, \mathsf{wp}\ \texttt{[f1/get]P1()}\ \{v.v = \mathsf{None} \vee \exists v'.\, v = \mathsf{Some}(v') * P(v')\}$
We can remove the $\square$-modality as our context is persistent. Let us now apply Löb induction to get the assumption

$$\triangleright\, \mathsf{wp}\ \texttt{[f1/get]P1()}\ \{v.v = \mathsf{None} \vee \exists v'.\, v = \mathsf{Some}(v') * P(v')\}$$

We will make use of this assumption in the case where we are forced to loop due to contention on the main stack. We next apply the bind rule to change our goal to

$$\mathsf{wp}\ \texttt{f1 ()}\ \{v.\mathsf{wp}\ (\texttt{match v with ...})\ \{v.v = \mathsf{None} \vee \exists v'.\, v = \mathsf{Some}(v') * P(v')\}\}$$

We can now apply the assumption that we have for f_1, namely we now must prove the entailment.

$$\begin{aligned}\forall v.\ &(v = \mathsf{None} \vee \exists v'.\, v = \mathsf{Some}(v') * P(v'))\\ &-\!\!*\ \mathsf{wp}\ (\texttt{match v with ...})\ \{v.v = \mathsf{None} \vee \exists v'.\, v = \mathsf{Some}(v') * P(v')\}\end{aligned}$$

Let us assume that we have some v and that $v = \mathsf{None} \vee \exists v'.\, v = \mathsf{Some}(v') * P(v')$ holds. We now case on this disjunction.

Let us first consider the case where $\exists v'.\, v = \mathsf{Some}(v') * P(v')$ holds. We now need to show can then destruct this existential telling us that there is some v' so that $v = \mathsf{Some}(v')$ and $P(v')$ holds. Rewriting by $v = \mathsf{Some}(v')$ we can then apply the β rule for matches to transform our goal into

$$\mathsf{wp}\ (\texttt{Some(v')})\ \{v.v = \mathsf{None} \vee \exists v'.\, v = \mathsf{Some}(v') * P(v')\}$$

However since this is a value it suffices to prove that

$$\exists v''.\, \mathsf{Some}(v'') = \mathsf{Some}(v') * P(v'')$$

however our assumptions give this immediately.

Now consider the case where $v = \mathsf{None}$. In this case we can rewrite again by this equality and apply the β rule for match to conclude our goal is

$$\mathsf{wp}\,\texttt{match !r with ...}\,\{v.v = \mathsf{None} \lor \exists v'.\, v = \mathsf{Some}(v') * P(v')\}$$

We now again apply our binding rule to rewrite our conclusion to the form

$$\mathsf{wp}\,(\texttt{!r})\,\{v.\mathsf{wp}\,\texttt{match v with ...}\,\{v.v = \mathsf{None} \lor \exists v'.\, v = \mathsf{Some}(v') * P(v')\}\}$$

Now that we have an atomic operation, we can apply our invariant rule to open the invariant that we have about r. Let us assume that we have $\triangleright\,\mathsf{stack\_inv}(r)$. We can then rewrite this to

$$\exists \ell, v'.\, \triangleright(v = \ell * \ell \mapsto v' * \mathsf{is\_stack}(v'))$$

which is of course equivalent to

$$\exists \ell, v'.\, \triangleright\, v = \ell * \triangleright\, \ell \mapsto v' * \triangleright\, \mathsf{is\_stack}(v')$$

For this, we then conclude that there exists some v' and ℓ so that these conditions hold. We then can step our goal to

$$\mathsf{wp}\,(\texttt{v'})\,\{v.\mathsf{wp}\,\texttt{match v with ...}\,\{v.v = \mathsf{None} \lor \exists v'.\, v = \mathsf{Some}(v') * P(v')\}\}$$

and remove the $\triangleright$s from our assumptions so that we have $v = \ell * \triangleright\, \ell \mapsto v' * \mathsf{is\_stack}(v')$. Next we unfold $\mathsf{is\_stack}(v')$ to conclude that either $v' = \mathsf{None}$ or there is a h, t so that $v = \mathsf{Some}(h, t)$ where $P(h)$ and $\triangleright\,\mathsf{is\_stack}(t)$ holds.

In the first case, we rewrite our goal to

$$\mathsf{wp}\,(\texttt{None})\,\{v.\mathsf{wp}\,\texttt{match v with ...}\,\{v.v = \mathsf{None} \lor \exists v'.\, v = \mathsf{Some}(v') * P(v')\}\}$$

Finally, we can easily reestablish our invariant as required since we have not consumed any resources, that is, we can prove $\triangleright\,\mathsf{stack\_inv}(r)$ using our assumptions that $v' = \mathsf{None}$ with $r = \ell$ and $\ell \mapsto v'$. Finally we now apply the β rule for match transforming our goal into

$$\mathsf{wp}\,\mathsf{None}\,\{v.v = \mathsf{None} \lor \exists v'.\, v = \mathsf{Some}(v') * P(v')\}$$

which is immediately established because $\mathsf{None} = \mathsf{None}$ obviously holds.

Let us consider the second case. We again rewrite by this equality and reestablish our invariant. We can reestablish our invariant because, again, we have just destructed it without consuming any of the resources it provided. Our goal is then

$$\mathsf{wp}\,\texttt{match Some(h, t) with ...}\,\{v.v = \mathsf{None} \lor \exists v'.\, v = \mathsf{Some}(v') * P(v')\}$$

Notice that because we packed up all of our knowledge of h, t back into our invariant, we have no information currently recorded about h or t. Next we apply the β rule for match as well as a few simple β reductions for projections to get

$$\mathsf{wp}\;\begin{array}{l}\texttt{if cas r (Some v') t}\\ \texttt{then Some h}\\ \texttt{else pop n}\end{array}\;\{v.v = \mathsf{None} \lor \exists v'.\, v = \mathsf{Some}(v') * P(v')\}$$

We then again apply our bind rule to change our goal to

$$\mathsf{wp}\,\texttt{cas r (Some v') t}\,\{v.\mathsf{wp}\,(\ldots)\,\{v.v = \mathsf{None} \lor \exists v'.\, v = \mathsf{Some}(v') * P(v')\}\}$$

We again open our invariant $\boxed{\mathsf{stack\_inv}(r)}^{\iota}$. This gives us that $\triangleright\,\mathsf{stack\_inv}(r)$. Unfolding all of this, we get that there is some ℓ' and some v''

$$r = \ell' * \triangleright\, \ell' \mapsto v'' * \triangleright\, \mathsf{is\_stack}(v'')$$

Let us then case on whether or not $v' = v''$.

In the first case we can successfully compute our CAS so our goal is

$$\mathsf{wp}\,(\texttt{if true then ... else ...})\,\{v.v = \mathsf{None} \vee \exists v'.\, v = \mathsf{Some}(v') * P(v')\}$$

where now $r = \ell'$ and $\ell' \mapsto t$. Since we have taken a step, we may strip the $\triangleright$s off of our assumption. We now turn to reestablishing our invariant which is nontrivial since we have changed what r points to. We must show that

$$\triangleright\, \exists \ell, v.\, r = \ell * \ell \mapsto v * \triangleright\, \mathsf{is\_stack}(v)$$

For this, we instantiate it with $\ell = \ell'$ and $v = t$. We then have the first two goals by assumption so we merely need to show that $\triangleright\, \mathsf{is\_stack}(t)$ holds. For this, we unfold our assumption that $\mathsf{is\_stack}(v'')$ holds. Since $v'' = \mathsf{Some}(h, t)$ we must have that

$$\exists h', t'.\, v'' = (h', t') * P(h') * \triangleright\, \mathsf{is\_stack}(t')$$

Unfolding this, we use injectivity to conclude that $h' = h$ and $t' = t$ so we have $P(h') * \triangleright\, \mathsf{is\_stack}(t')$. The latter gives us immediately what we need to reestablish the invariant. We also hold on to the assumption that $P(h)$ holds and return to our goal that

$$\mathsf{wp}\ \begin{array}{l}\texttt{if true}\\ \texttt{then Some(h)}\\ \texttt{else pop(n)}\end{array}\ \{v.v = \mathsf{None} \vee \exists v'.\, v = \mathsf{Some}(v') * P(v')\}$$

so we step this to conclude that we must show

$$\mathsf{Some}(h) = \mathsf{None} \vee \exists v'.\, \mathsf{Some}(h) = \mathsf{Some}(v') * P(v')$$

however the right disjunct of this holds by assumption so we're done with this case.

Finally, we consider the case that $v' \neq v''$. In this case our CAS fails so we can step our goal to

$$\mathsf{wp}\ \begin{array}{l}\texttt{if false}\\ \texttt{then Some(h)}\\ \texttt{else pop(n)}\end{array}\ \{v.v = \mathsf{None} \vee \exists v'.\, v = \mathsf{Some}(v') * P(v')\}$$

it is trivial to reestablish our invariant since we know that $\mathsf{stack\_inv}(v'')$ still holds as we have not consumed any of the resources. Finally we apply the β rule for if to turn out goal into

$$\mathsf{wp}\,\texttt{pop(n)}\,\{v.v = \mathsf{None} \vee \exists v'.\, v = \mathsf{Some}(v') * P(v')\}$$

but now we apply our IH that we created earlier using Löb induction and we're done.

(2) $\forall v.\, P(v) \mathrel{-\!\!*} \mathsf{wp}\ \texttt{[f1/get]P2(v)}\ \{\mathsf{True}\}$
This is left as an exercise to the reader as it is strictly simpler than the above proof.

□

The Coq formalization of all of this can be found in `concurrent_stack2.v`.

4. A Second Formalization: View-Shifts without Helping

The specification proven above has a major defect, it doesn't express any sort of ordering on the underlying collection of objects. Informally, it is quite clear that something popped out of the stack must have been pushed in at some point. This can be argued by a parametricity-type argument about P. Nothing stops us, however, from ascribing this specification to a concurrent queue or even something that permutes its elements randomly!

In a concurrent setting this may seem like the best possible specification though. After all, suppose we have the following code.

```
push(1);
push(2);
let x = pop () in
let y = pop () in
(x, y)
```

It is perfectly possible for this code to evaluate to either $(1, 2)$ or $(2, 1)$ or even $(6, 7)$. This is because another thread could come along between the time when the `push`s have been executed, perform an arbitrary combination of `push`s and `pop`s on the stack before the original thread can execute its two `pop`s. There is a far more serious flaw in the specification though: it is satisfied by operations which always return `None` and discard the input!

What is needed is a specification that reflects the fact that there is an underlying stack that may be accessed atomically and the behavior of the function is determined by the state of this stack at some particular point. This is related to the standard concurrency idea of *linearization* where a complicated operation can be reduced to a single atomic interaction. This means that several complex, overlapping concurrent operations on a data structure can be logically linearly ordered.

To begin experimenting with this new form of specification, we will start by verifying a concurrent stack without helping. This simplifies the verification considerably. Since, furthermore, helping is an invisible optimization any good specification for a stack with helping is an equally good specification for a stack without helping.

In order to do define the specification we shift from parameterizing our specifications from properties of elements to properties of the abstract stack that our data structure represents. This abstract stack can be represented with a simple list at the level of the logic. It will then be assumed that Iris is supplemented with lists of values, a basic elimination operation on them (called foldr), and the basic rules of equality for it.

Turning now to the question of what specifications ought to be concretely, they will have to make use of *view-shifts*. This is a feature of the Iris base logic which generalizes simple implication, $-\!\!*$, to allow for the updating of ghost state and the usage of invariants. These view-shifts are useful in defining the specification because these precisely capture the idea of a linearization point. If we supply an operation with a viewshift $\forall L.\, P(L) \Rrightarrow\!\!\!\!*_{\mathcal{E}}\; P(L) * Q$

and that operation is equipped with the specification[2]

$$\boxed{P(L)}^N \twoheadrightarrow \mathsf{wp}\ ...\ \{Q\}$$

then at some point in the code, for an atomic operation the invariant will be opened and the view-shift will be used. This must happen because it is the only way to produce a Q to complete this specification. This view-shift then isolates the reasoning that will be done during that atomic step using the proposition held by the invariant. Furthermore, affinity means that such reasoning can only be applied once. This interaction with the proposition isolated by the invariant is precisely a linearization point; it's the only time which we manipulate the shared state guarded by the invariant. These specifications provide a much more flexible way of interacting with concurrent functions because they, in effect, capture any specification which contains the same critical element of

(1) A single abstract property of the stack represented by the data-structure.
(2) The post conditions are implied (with the possible manipulation of ghost state) by the state of abstract data structure at the moment it is accessed.

The specification for a push operation might look like this

$$\forall v.\, (\forall vs.\, P(vs) \Rrightarrow\!\!\!\!\!\!\ast_{\top \setminus N^\uparrow}\ Q * P(v :: vs)) \twoheadrightarrow \mathsf{wp}\ \texttt{push(v)}\ \{v.Q\}$$

The $\setminus N^\uparrow$ can be safely ignored for now.[3] In other words, if given a means of atomically switching from $P(vs)$ to $Q * P(v :: vs)$ we can produce a Q in our post condition. If we want to specialize this to what we had earlier, then $Q = \mathsf{True}$ and $P(L) = \forall x \in L.\, P(x)$ gives us the specification we used to have for `push`. The real advantage of this style of specification is that we can express a lot more than this though. For an example of how this might be done, see Svendsen et al. [2013] which uses these specification to automatically derive precise *sequential* specifications from the concurrent ones without undue effort! It is of course a balance to create an expressive and yet general specification but the fact that this specification can encapsulate both the sequential case as well as the highly concurrent bag-like specification is evidence for its utility.

Turning now to the question of specifying the invariants and predicates we need to make these specification work, we will need a new definition of $\mathsf{is\_stack}$ and $\mathsf{stack\_inv}$. The idea is that the stack invariant will contain that there is some stack in a mutable cell representing an abstract stack vs so that $P(vs)$ holds. That is,

$$\begin{aligned}
\mathsf{is_stack}([], v) &\triangleq v = \mathsf{None} \\
\mathsf{is_stack}(x :: L, v) &\triangleq v = \exists t.\, v = \mathsf{Some}(x, t) * \mathsf{is_stack}(L, t) \\
\mathsf{stack_inv}(v) &\triangleq \exists \ell, v', L.\, v = \ell * \ell \mapsto v' * P(L) * \mathsf{is_stack}(L, v')
\end{aligned}$$

Here we have parameterized our construction by the property we maintain about the abstract stack, P. Let us further assume that we have some arbitrary Q, Q_1, and Q_2 which will parameterize our specification. They are specified up front simply to avoid the tedium of writing quantifiers for them over and over again but no restrictions need to be imposed

[2]In this section and onwards we are going to use invariant namespaces instead of concrete invariant names. The reason for this will become clear soon. It is defined as $\boxed{P(L)}^N = \exists \iota \in N.\boxed{P(L)}^\iota$.

[3]For the curious reader: it means that the view shift can open every invariant but invariants in the upwards-closure of the namespace N.

on them. By varying P, along with Q, Q_1 and Q_2 it is possible to recover our original specification as a concurrent bag amongst others.

Therefore, the specification implies that we take the $P(L)$ contained in the invariant and atomically update it to $P(x :: L)$ before rebundling the whole thing back into the $\mathsf{stack\_inv}$. The full specification for the stack data structure again makes use of the "cut" trick seen earlier.

(3)

$$
\begin{aligned}
&\forall \Phi.\\
&\quad (\forall f_1 f_2.\\
&\qquad (\square(((\forall vs.\, P(v :: vs) \Rrightarrow[\top \setminus N^{\uparrow}] Q_1(v) * P(vs)) \wand (\forall vs.\, P([]) \Rrightarrow[\top \setminus N^{\uparrow}] Q_2 * P([])) \wand\\
&\qquad\quad \mathsf{wp}\ f_1()\,\{v.v = \mathsf{None} * Q_2 \vee \exists v'.\, v = \mathsf{Some}(v') * Q_1(v')\})\\
&\qquad\quad \wand \square(\forall v.\, (\forall vs.\, P(vs) \Rrightarrow[\top \setminus N^{\uparrow}] Q * P(v :: vs)) \wand \mathsf{wp}\ f_2(v)\,\{v.Q\})\\
&\qquad\quad \wand\ \ \Phi(f_1, f_2))\\
&\qquad \wand P([])\\
&\qquad \wand \mathsf{wp}\ \mathsf{stack}()\,\{\Phi\}
\end{aligned}
$$

[4]

The code that will be verified is slightly modified from the previous version. It is shown in Figure 5

Theorem 4.1. *Proposition (3) holds.*

Proof. As before, we start by stepping our program, leaving us with the goal

$$\mathsf{wp}\ \texttt{let r = ref None in ...}\,\{\Phi\}$$

with the assumption

$$
\begin{aligned}
&\forall f_1 f_2.\\
&\quad (\square(((\forall vs.\, P(v :: vs) \Rrightarrow[\top \setminus N^{\uparrow}] Q_1(v) * P(vs)) \wand (\forall vs.\, P([]) \Rrightarrow[\top \setminus N^{\uparrow}] Q_2 * P([])) \wand\\
&\quad\quad \mathsf{wp}\ f_1()\,\{v.v = \mathsf{None} * Q_2 \vee \exists v'.\, v = \mathsf{Some}(v') * Q_1(v')\})\\
&\quad\quad \wand \square(\forall v.\, (\forall vs.\, P(vs) \Rrightarrow[\top \setminus N^{\uparrow}] Q * P(v :: vs)) \wand \mathsf{wp}\ f_2(v)\,\{v.Q\})\\
&\quad\quad \wand\ \ \Phi(f_1, f_2))
\end{aligned}
$$

as well as $P([])$. We then apply the application rule giving us some ℓ so that $\ell \mapsto \mathsf{None}$. We take this time to establish the stack invariant for r in the namespace N, that is $\boxed{\mathsf{stack\_inv}(r)}^{N}$. In order to do this we must show that

$$\triangleright(\exists \ell', v, L.\, \ell = \ell' * \ell' \mapsto v * P(L) * \mathsf{is\_stack}(L, v))$$

[4]The reason for using namespaces is, when allocating invariants we can choose the namespace N in which we allocate it. We can thus specify that the client provided view shifts may not depend on the invariant, by saying that they are not allowed to open any invariants in $N^{\uparrow}$. This is needed for the verification of `pop`.

Had we on the other hand used the invariant allocation rule for concrete invariant names, then the invariant name we would get is existentially quantified. We would thus only be able to state, that there exists an invariant that the client provided view shifts may not depend on. This would of course make it impossible for the view shifts to depend on any invariants as the client does not know, which particular invariant he/she can not open.

```
stack = fun () →
  let r = ref None in
  (rec pop n →
     match !r with
        None → None
      | Some hd =>
        if cas r (Some hd) (snd hd)
        then Some (fst hd)
        else pop n
      end,
    rec push n →
      let r' = !r in
      let r'' = Some (n, r') in
      if cas r r' r''
      then ()
      else push n)
```

FIGURE 5. Concurrent stack *without* helping.

holds according to the invariant allocation rule.

We will prove this for ℓ, None, and [] respectively. We then merely must show that the following holds.

$$P([]) * \mathsf{is\_stack}([], \mathsf{None})$$

However the left-side of this conjunction is an assumption and the right-hand side is simply true by reflexivity. We then, returning to our main goal, need to show that

$$\mathsf{wp}\,(\ldots,\ \ldots)\,\{\Phi\}$$

holds, and for this we apply our assumption. This leaves us with two goals, showing that our specification holds for push and pop. We will consider only the case for `push` as an illustrative example. That is, we want to show for an arbitrary that v

$$\Box(\forall L.\, P(L) \Rrightarrow\!\!\!\!\!\!\ast_{\top\setminus N\uparrow}\ P(v :: L) * Q) \mathrel{-\!\!*}$$
$$\mathsf{wp}\,(\texttt{(rec push n -> let r' = !r in let r'' = Some (n, r') in ...)}\ \ \texttt{v})\,\{Q\}$$

We can easily get rid of the $\Box$-modality, as our context is persistent. We then take a moment to use Löb induction i.e. we may assume

$$\triangleright(\forall v, L.\, P(L) \Rrightarrow\!\!\!\!\!\!\ast_{\top\setminus N\uparrow}\ P(v :: L) * Q) \mathrel{-\!\!*}$$
$$\mathsf{wp}\,(\texttt{(rec push n -> let r' = !r in let r'' = Some (n, r') in ...)}\ \ \texttt{v})\,\{Q\}$$

We can now use the β-rule as well as the bind-rule, hence it suffices to show

$$\mathsf{wp}\,\texttt{!r}\,\{v'.\mathsf{wp}\,(\texttt{let r' = v' in let r'' = Some (v, r') in ...})\,\{Q\}\}$$

At this point we open the invariant $\boxed{\mathsf{stack\_inv}(r)}^N$. This gives us that there is some ℓ, v'' and L so that

$$\triangleright(r = \ell * \ell \mapsto v'' * P(L) * \mathsf{is\_stack}(L, v''))$$

Using this, we can step our goal to

$$\mathsf{wp}\,(\texttt{let r' = v'' in let r'' = Some (v, r') in ...})\,\{Q\}$$

and re-establish our invariant trivially since we have not consumed any resources. In this position we can step our goal to

$$\mathsf{wp}\,(\texttt{if cas r v'' (Some (v, v'')) then () else push v})\,\{Q\}$$

At this point we open up our invariant again, giving us

$$\triangleright(r = \ell * \ell \mapsto v''' * P(L) * \mathsf{is\_stack}(L, v'''))$$

for some v'''. We then case on whether or not $v'' = v'''$.

First consider the case that this does hold. In this case, we can step our CAS successfully giving us the new assumption that $\ell \mapsto \mathsf{Some}(v, v'')$ and the obligation to reestablish our invariant and show that Q holds. For this we first apply

$$P(L) \Rrightarrow\!\!\!\!\ast_{\top\setminus N\uparrow}\ P(v :: L) * Q$$

to our assumption that $P(L)$ holds (we may strip of the later since we have taken a step of computation). This gives us that $\Rrightarrow_{\top\setminus\iota} P(v :: L) * Q$ holds. We then have that $\mathsf{is\_stack}(v :: L, \mathsf{Some}(v, v''))$ holds because we have assumed that $\mathsf{is\_stack}(L, v'')$ holds with $v'' = v'''$. This, combined with our assumption that $\ell \mapsto \mathsf{Some}(v, v'')$ gives us that $\Rrightarrow_{\top\setminus N\uparrow} \mathsf{stack\_inv}(r)$ holds so we may can re-establish our invariant with using Q which we then use to discharge the remaining postcondition.

If instead $v'' \neq v'''$ then we can reduce our goal to

$$\mathsf{wp}\,(\texttt{if false then () else push v})\,\{Q\}$$

and immediately re-establish our invariant because we have not changed anything. But then stepping and applying our induction hypothesis immediately gives us the desired conclusion. □

The Coq formalization of this specification and proof may be found in `concurrent_stack3.v`.

5. A Third Formalization: View-Shifts with Helping

It is a straightforward exercise to adapt the previous proof to work with helping using the same setup as the previous proof. Indeed, since helping is an invisible optimization to a concurrent data structure the same specification ought to apply. The only question is how to modify the specifications we give to mailboxes in order to suitably handle these view-shifts.

The primary difference is in the definition of the invariants for offers and mailboxes. Rather than having them pass around ownership of $P(v)$ for some P where v is the value they contain, they have to pass around the right to perform certain view-shifts. This unfortunately means that the proofs for mailboxes and offers become entangled with that of stacks to a much larger degree. This is why the Coq development does not isolate them into separate lemmas any more.

The new invariant for offers is now defined using

$$\begin{aligned}\mathsf{stages}_\gamma(\ell, v) \triangleq &(\ell \mapsto 0 * (\forall vs.\, P(vs) \mathrel{-\!\!*\!\!\!\Rightarrow}_{\top\setminus N^\uparrow} P(v :: vs) * Q)) \\ &\vee (\ell \mapsto 1 * Q) \\ &\vee (\ell \mapsto 1 * \boxed{\mathsf{ex}(())}^{\gamma}) \\ &\vee (\ell \mapsto 2 * \boxed{\mathsf{ex}(())}^{\gamma})\end{aligned}$$

This means that either one of two interactions is possible

(1) A thread without ownership of $\boxed{\mathsf{ex}(())}^{\gamma}$ attempts to switch from $\ell \mapsto 0$ to $\ell \mapsto 1$. In doing so, it is obligated to take the view-shift $\forall vs.\, P(vs) \mathrel{-\!\!*\!\!\!\Rightarrow}_{\top\setminus N^\uparrow} P(v :: vs) * Q$ and place the resulting Q back in the invariant.
(2) The thread which owns $\boxed{\mathsf{ex}(())}^{\gamma}$ may attempt to revoke this offer, switching it from $\ell \mapsto 0$ to $\ell \mapsto 2$. If it is successful then it can return the view-shift transitioning $\forall vs.\, P(vs) \mathrel{-\!\!*\!\!\!\Rightarrow}_{\top\setminus N^\uparrow} P(v :: vs) * Q$ or it fails, implying that some thread has already successfully executed a take. If the latter is the case, the thread can return the Q that must be stored in stages which must be in the disjunct $\ell \mapsto 1 * Q$.

With this we can specify the invariant enforced on offers.

$$\mathsf{is\_offer}_\gamma(v) \triangleq \exists v' l.\, v = (v', l) * \boxed{\mathsf{stages}_\gamma(l)}^{N.offerInv}$$

The only difference worth noting is that the existentially quantified invariant name does now belong to the namespace *N.offerInv*. This is purely an artifact of the particular approach the proof of the specification has taken. At various points, it will be necessary to simultaneously open the invariant containing knowledge of the stack as well as the invariant containing what stage an offer is in. For instance, this is how `take_offer` will work so that it can actually apply the view-shift to the current state of the stack. Opening two invariants at the same time, however, is only possible if the invariants are in fact disjoint. Hence if we allocate them in different namespaces, we know they are disjoint for free. The reason for using sub-namespaces of N is, that we can easily specify that a client may not depend on any of the invariants, by stating that he/she may not depend on any invariants in $N^\uparrow$.

Rather than proceeding to formalize the specifications of mailboxes and offers, it is simpler to proceed directly to formalizing the stack data structure itself. The invariant governing a stack is unchanged from the previous section.

$$\begin{aligned}\mathsf{is\_stack}([], v) &\triangleq v = \mathsf{None} \\ \mathsf{is\_stack}(x :: L, v) &\triangleq v = \exists t.\, v = \mathsf{Some}(x, t) * \mathsf{is\_stack}(L, t) \\ \mathsf{stack\_inv}(v) &\triangleq \exists \ell, v', L.\, v = \ell * \ell \mapsto v' * P(L) * \mathsf{is\_stack}(L, v')\end{aligned}$$

The final specification is again unchanged. For brevity, the proof (which is largely a combination of the two previously explained ones) is only sketched.

Theorem 5.1. *The following specification holds.*

$$
\begin{aligned}
&\forall \Phi.\\
&\quad (\forall f_1 f_2.\\
&\qquad (\square(((\forall vs.\, P(v :: vs) \Rrightarrow[\top \setminus N^{\uparrow}] Q_1(v) * P(vs)) \wedge (\forall vs.\, P([]) \Rrightarrow[\top \setminus N^{\uparrow}] Q_2 * P([])) \mathrel{-\!\!*}\\
&\qquad\quad \mathit{wp}\, f_1()\, \{v.v = \mathsf{None} * Q_2 \vee \exists v'.\, v = \mathsf{Some}(v') * Q_1(v')\})\\
&\qquad\quad \mathrel{-\!\!*} \square(\forall v.\, (\forall vs.\, P(vs) \Rrightarrow[\top \setminus N^{\uparrow}] Q * P(v :: vs)) \mathrel{-\!\!*} \mathit{wp}\, f_2(v)\, \{v.Q\})\\
&\qquad\quad \mathrel{-\!\!*}\ \Phi(f_1, f_2))\\
&\qquad \mathrel{-\!\!*} P([])\\
&\qquad \mathrel{-\!\!*} \mathit{wp}\, \mathit{stack}()\, \{\Phi\}
\end{aligned}
$$

Proof. The beginning of the proof is a straightforward combination of the previous two proofs, allocating an invariant for the mailbox and the mutable cell containing the stack: $\boxed{\mathsf{stack\_inv}(r)}^{N.stackInv}$ and $\boxed{\mathsf{is\_mailbox}(r')}^{N.mailboxInv}$.

Having done this, we apply the assumption that

$$
\begin{aligned}
&\forall f_1 f_2.\\
&\qquad (\square(((\forall vs.\, P(v :: vs) \Rrightarrow[\top \setminus N^{\uparrow}] Q_1(v) * P(vs)) \wedge (\forall vs.\, P([]) \Rrightarrow[\top \setminus N^{\uparrow}] Q_2 * P([])) \mathrel{-\!\!*}\\
&\qquad\quad \mathsf{wp}\, f_1()\, \{v.v = \mathsf{None} * Q_2 \vee \exists v'.\, v = \mathsf{Some}(v') * Q_1(v')\})\\
&\qquad\quad \mathrel{-\!\!*} \square(\forall v.\, (\forall vs.\, P(vs) \Rrightarrow[\top \setminus N^{\uparrow}] Q * P(v :: vs)) \mathrel{-\!\!*} \mathsf{wp}\, f_2(v)\, \{v.Q\})\\
&\qquad\quad \mathrel{-\!\!*}\ \Phi(f_1, f_2))
\end{aligned}
$$

This leaves us with two separate verifications for push and pop. These are largely the same as the previous proof. The only difference is each proof is proceeded by manual manipulations of the mailboxes and offers which in turn are simply inlined versions of the proof from the prior sections.

We will consider the case for `push`. The goal we must prove is then

$$\square(\forall v.\, (\forall vs.\, P(vs) \Rrightarrow[\top \setminus N^{\uparrow}] Q * P(v :: vs)) \mathrel{-\!\!*} \mathsf{wp}\, \texttt{push(v)}\, \{v.Q\})$$

As our context just consist of invariants and therefore is persistent, we may remove the $\square$-modality. Now, let us suppose that $\forall vs.\, P(vs) \Rrightarrow[\top \setminus N^{\uparrow}] Q * P(v :: vs)$ and that we have some value v. It as at this point that we apply Löb induction to add an induction hypothesis of $\triangleright\, \mathsf{wp}\, \texttt{push(v)}\, \{v.Q\}$ to our context. We then simplify our goal using β and bind to

$$\mathsf{wp}\, \texttt{put v}\, \{v'.\mathsf{wp}\, \texttt{match v' with ...}\, \{Q\}\}$$

Since we have no specification yet formalized for `push`, we proceed to unfold it and begin to work with the internal implementation of `put`.

$$\mathsf{wp}\, \texttt{let off = mk\_offer v in ...}\, \{v'.\mathsf{wp}\, \texttt{match v' with ...}\, \{Q\}\}$$

We can simplify this again and apply bind

$$\mathsf{wp}\, \texttt{(v, ref 0)}\, \{v'.\mathsf{wp}\, \texttt{let off = v' in ...}\, \{v'.\mathsf{wp}\, \texttt{match v' with ...}\, \{Q\}\}\}$$

We can then apply the rule for allocation, so suppose that we have some ℓ and $\ell \mapsto 0$. We then must show that

$$\mathsf{wp}\, \texttt{r' := Some (v, l); revoke\_offer (v, l)}\, \{v'.\mathsf{wp}\, \texttt{match v' with ...}\, \{Q\}\}$$

We may then open the invariant $\boxed{\mathsf{is\_mailbox}(r')}^{N.mailboxInv}$ and replace the previous value stored in r' with `(v, l)` and re-establish the invariant $\boxed{\mathsf{is\_mailbox}(r')}^{N.mailboxInv}$. In order to do this, we must use $\forall vs.\, P(vs) \Rrightarrow\!\!\!\!\ast_{\top\setminus N\uparrow}\ Q * P(v :: vs)$[5] and allocate a new invariant $\boxed{\mathsf{stages}_\gamma(l)}^{N.offerInv}$ for some γ so that we also have $\boxed{\mathsf{ex}(())}^\gamma$. Thus, our goal is now

$$\mathsf{wp}\,\texttt{revoke\_offer (v, l)}\,\{v'.\mathsf{wp}\,\texttt{match v' with ...}\,\{Q\}\}$$

If we apply β rules we then end up with

$$\mathsf{wp}\,(\texttt{if cas l 0 2 then Some(v) else None})\,\{v'.\mathsf{wp}\,\texttt{match v' with ...}\,\{Q\}\}$$

Let us then open up the invariant $\boxed{\mathsf{stages}(l)}^{N.offerInv}$. This tells us that

$$\begin{aligned}&(\ell \mapsto 0 * (\forall vs.\, P(vs) \Rrightarrow\!\!\!\!\ast_{\top\setminus N\uparrow}\ P(v :: vs) * Q))\\ \vee\,&(\ell \mapsto 1 * Q)\\ \vee\,&\ell \mapsto 1 * \boxed{\mathsf{ex}(())}^\gamma\\ \vee\,&(\ell \mapsto 2 * \boxed{\mathsf{ex}(())}^\gamma)\end{aligned}$$

Let us perform case analysis on these three possible results. We can immediately dismiss the last two cases since we have assumed to own $\boxed{\mathsf{ex}(())}^\gamma$ which means that the $\boxed{\mathsf{ex}(())}^\gamma$ coming from this invariant would produce a contradiction.

In the second case, the CAS fails so we are left with $\ell \mapsto 1$, $\boxed{\mathsf{ex}(())}^\gamma$, and Q. Let us re-establish our invariant using the $\ell \mapsto 1$ and $\boxed{\mathsf{ex}(())}^\gamma$ using the second branch. Our goal is then

$$\mathsf{wp}\,(\texttt{None})\,\{v'.\mathsf{wp}\,\texttt{match v' with ...}\,\{Q\}\}$$

and simplifying this gives us the obligation

$$\mathsf{wp}\,\texttt{None}\,\{Q\}$$

but this is trivial to discharge using our assumption of Q that we got from the invariant. This case is essentially the position we would be in if the side-channel is accepted.

In the first case, therefore, we must consider what happens if the side-channel is not used. Clearly the CAS is successful and we can re-establish this invariant by providing $\boxed{\mathsf{ex}(())}^\gamma$ and choosing the last disjunct. In this case, we have the assumption $\forall vs.\, P(vs) \Rrightarrow\!\!\!\!\ast_{\top\setminus N\uparrow}$ $P(v :: vs) * Q$ and the goal

$$\mathsf{wp}\,(\texttt{Some(v)})\,\{v'.\mathsf{wp}\,\texttt{match v' with ...}\,\{Q\}\}$$

Therefore, we can apply β rules to reduce this to

$$\mathsf{wp}\,\texttt{let x = !r in let y = Some (v, x) in ...}\,\{Q\}$$

In order to reduce this, we must open the invariant $\boxed{\mathsf{stack\_inv}(r)}^{N.stackInv}$ which tells us, amongst other things, that $r = \ell'$ for some ℓ' and that $\ell' \mapsto s$ for some value s. We can then step our program and re-establish the invariant (stepping does not effect the context at all) to

$$\mathsf{wp}\,\texttt{let x = s in let y = Some (v, x) in ...}\,\{Q\}$$

[5]Here we use, that the view-shift does not open the mailbox invariant

We can now repeatedly simplify our goal to

$$\texttt{wp if cas r s (Some (v, s)) then () else push v}\,\{Q\}$$

In order to simplify this, we must once again open up the invariant $\boxed{\mathsf{stack\_inv}(r)}^{N.stackInv}$. This tells us that for some s', L and ℓ' that

$$r = \ell' * \ell' \mapsto s' * P(L) * \mathsf{is\_stack}(L, s')$$

holds. Let us then case on whether or not $s' = s$.

If it does, then the CAS succeeds, therefore we have $\ell' \mapsto \mathsf{Some}(v, s)$. Simple logic gives us that $\mathsf{is\_stack}(L, s')$ implies that $\mathsf{is\_stack}(v :: L, \mathsf{Some}(v, s))$ holds. In order to re-establish our invariant we must show that $\Rrightarrow \mathsf{stack\_inv}(r)$ holds. However, since we have the implication $P(L) \Rrightarrow\!\!\!\!\ast_{\top \setminus N\uparrow} P(v :: L)$ it is sufficient to show that

$$r = \ell' * \ell' \mapsto \mathsf{Some}(v, s) * P(L) * \mathsf{is\_stack}(v :: L, \mathsf{Some}(v, s))$$

however we have precisely these assumptions so we're done. This leaves us with the goal $\mathsf{wp}\,()\,\{Q\}$ which is discharged immediately with our assumption of Q.

If these are not equal, then the CAS fails so it's trivial to re-establish the invariant. Our goal is then $\mathsf{wp}\,\texttt{push v}\,\{Q\}$ but for this we just apply our IH and we're done. □

The Coq formalization of this specification and proof may be found in `concurrent_stack4.v`.

6. Conclusion

In this case study we have examined several different incarnations of formalizations of concurrent stacks in Iris. This provides evidence for Iris being an expressive and flexible program logic. Several of Iris's features were necessary to even express the desired specifications.

- Impredicative invariants were needed in order to have the invariant contain P, the arbitrary predicate all the specifications where parameterized by.
- Higher-order specifications in order to describe the *closure-returning* pattern that mailboxes and stacks made use of.
- View-shifts in order to express linearization points.

Furthermore, the encoding of state transition systems as a simple proposition using ghost state demonstrates how simple CMRAs are sufficient to encode complex logical structures for expressing the structure of our program.

All of these specifications where heavily inspired by Clausen [2017] which provided a similar verification of hash-tables in Iris. Future work in this direction would be to mimic this work and drive towards more compositional verification of concurrent stacks. Ideally, the proof could be decomposed in the same that the proof of the bag specification is: respecting abstraction boundaries of APIs and relying purely on the specifications. More generally, there is still a great deal of engineering as well as theoretical to work in specifying sophisticated data-structures in a useful but still provable way.

References

Esben Galvind Clausen. Verifying Hash Tables in Iris. Master's thesis, Aarhus University, 2017.

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2015.

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Higher-order ghost state. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, 2016.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the Ground Up: A Modular Foundation for Higher-Order Concurrent Separation Logic. Preprint., 2017. URL `http://www.mpi-sws.org/ dreyer/papers/iris-ground-up/paper.pdf`.

Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. *The Essence of Higher-Order Concurrent Separation Logic*. 2017a.

Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive Proofs in Higher-order Concurrent Separation Logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, 2017b.

K. Svendsen and L. Birkedal. Impredicative Concurrent Abstract Predicates. In *Programming Languages and Systems, 23rd European Symposium on Programming*, 2014.

K. Svendsen, L. Birkedal, and M. Parkinson. Modular reasoning about separation of concurrent data structures. In *Programming Languages and Systems, 22nd European Symposium on Programming*, 2013.