RELOC RELOADED: TECHNICAL APPENDIX

DAN FRUMIN, ROBBERT KREBBERS, AND LARS BIRKEDAL

Contents

1. Linearizability of stack with helping

As we have mentioned in the main body of the paper, the rules of ReLoC are not complete, and it particular data structures that use *helping* cannot be handled in ReLoC completely. However, by temporarily breaking the ReLoC abstraction, we can still verify such examples.

In this section we prove that a concurrent stack with helping refines a coarse-grained concurrent stack. For verifying this example we need to unfold the definition of relational judgment and work inside the model. Luckily, we do not have to give up on reasoning with ReLoC rules completely. Rather, we can still use ReLoC rules for carrying out a part of the proof. The goal of this section is to demonstrate that even for proofs that cannot be completed without breaking the ReLoC abstract we can still keep the compositionality of the overall proof, by isolating and encapsulating the bits that require breaking the abstraction, and providing suitable specifications for those bits.

The main theorem that we show in this section is the linearisability proof of a concurrent stack data structure. It is a fine-grained concurrent stack, in which two threads doing push and pop operations concurrently can synchronize and exchange data without touching the stack itself. Whenever a thread tries to push an element x on top of the stack, it first *offers* x. If there is a concurrent thread doing a pop operation, then it can *help* the first thread by taking up its offer. If no thread takes up the offer before it is withdrawn, then the element x is put on top of the stack (in a thread-safe way). The data structure is taken from the Iris lecture notes, where the authors give stacks with helping a bag-like specification in [BB20, Chapter 8]. It is a simplified version of the elimination-backoff stack from [HSY04], where the backoff array is of size 1.

We wish to verify that a stack with helping refines a coarse-grained stack, which uses a sequential stack and locks to guarantee mutual exclusion. We start be describing the data structure used for managing offers (Section 1.1) and the implementation of stack with helping (Section 1.2). We then present the coarse-grained stack which we use as a specification (Section 1.3) and the associated rules. Finally, we give the specification

Date: June 24, 2020.

for offers (Section 1.4), which we prove by breaking the ReLoC abstraction, and use that specification to prove that stack with helping refines the coarse-grained stack (Section 1.5).

1.1. **Offers.** In order to facilitate helping, we use an abstraction of *offers*, which are that can be created, to store a specific value; accepted; or revoked:

$$\exists \texttt{offer}. \begin{cases} \mathit{mk\_offer} & : \alpha \to \texttt{offer} \\ \mathit{revoke\_offer} & : \texttt{offer} \to \alpha\ \mathsf{option} \\ \mathit{accept\_offer} & : \texttt{offer} \to \alpha\ \mathsf{option} \end{cases}$$

The option type is defined as $\tau\ \mathsf{option} \triangleq \tau + \mathsf{unit}$, and its constructors are $\textbf{Some}(e) \triangleq \mathsf{inl}(e)$ and $\textbf{None} \triangleq \mathsf{inr}(())$.

If an offer was already accepted or revoked, then *revoke_offer* and *accept_offer* return **None**; otherwise those functions return the value associated with an offer. We implement an offer as a pair of a value and a reference, on which we synchronize the state of the offer:

$$\begin{aligned} \mathit{mk\_offer}\ v &= (v, \mathsf{ref}(0)) \\ \mathit{revoke\_offer}\ (v, \ell) &= \mathsf{if}\ \mathsf{CAS}(\ell, 0, 2)\ \mathsf{then}\ \textbf{Some}(v)\ \mathsf{else}\ \textbf{None} \\ \mathit{accept\_offer}\ (v, \ell) &= \mathsf{if}\ \mathsf{CAS}(\ell, 0, 1)\ \mathsf{then}\ \textbf{Some}(v)\ \mathsf{else}\ \textbf{None} \end{aligned}$$

An offer can be in three states: initially offered (0), accepted (1) and revoked (2). For our purposes the only thread that can revoke an offer is the one that put it up in the first place.

1.2. **Stack implementation.** The implementation of the stack with helping is given in Figure 1. The stack with helping with use a single location *mb* for managing the offers, which we call a *mailbox*. This mailbox is a part of the internal state of the stack. The other internal state is the stack contents itself, represented in memory as a linked list in the location r.

The internal representation of stack contents can be described as the following mutually recursive data type:

$$\begin{aligned} \texttt{node} &= \Lambda\alpha.(\mathsf{ref}\ (\alpha \times \texttt{stack}\ \alpha))\ \mathsf{option} \\ \texttt{stack} &= \Lambda\alpha.\mathsf{ref}\ (\texttt{node}\ \alpha) \end{aligned}$$

or in OCaml notation:

$$\begin{aligned} &\mathsf{type}\ \alpha\ \texttt{node} = (\alpha \times \alpha\ \texttt{stack})\ \texttt{ref option} \\ &\mathsf{and}\ \alpha\ \texttt{stack} = \alpha\ \texttt{node ref} \end{aligned}$$

In words: a stack is a reference to a node; a node is either **None** (representing the end of the stack), or a $\textbf{Some}(p)$ for a reference p that contains a pair $(\mathit{hd}, \mathit{tail})$ where hd is a element of the stack and *tail* is a stack.

Initially, the stack r is empty, and the mailbox *mb* is unset – both point to **None**.

The push operation initially does not modify the internal contents of the stack, but first creates a new offer*off*, which is shared with other threads through the mailbox location *mb*. Immediately after sharing *off*, the offer is revoked. If *revoke_offer off* returns **None**, then some other thread has happened to accept the offer in-between the two operations. In that case the push operation simply terminates. If *revoke_offer off* returns $\textbf{Some}(v)$ then no other thread has managed to accept the offer and *off* was successfully revoked. In that case the actual push happens on the underlying stack r.

The pop operation similarly uses the side channel *mb*: it first checks if there is an offer in the mailbox. If no offer is present, it resorts to popping an element off the stack r. Otherwise, it tries to accept the offer with *accept_offer off*. That operation can fail as well, in which case it resorts to calling *pop_no_offer* again.

$$
\begin{aligned}
mk\_stack() = {} & \mathsf{let}\ r = \mathsf{ref}(\mathbf{None})\ \mathsf{in} \\
& \mathsf{let}\ mb = \mathsf{ref}(\mathbf{None})\ \mathsf{in} \\
& (\lambda().\ pop\ r\ mb, \lambda x.\ push\ r\ mb\ x) \\
push\ r\ mb\ v = {} & \mathsf{let}\ off = mk\_offer\ v\ \mathsf{in} \\
& mb \leftarrow \mathbf{Some}(off); \\
& \mathsf{match}\ revoke\_offer\ off\ \mathsf{with} \\
& \mid \mathbf{None} \rightarrow () \\
& \mid \mathbf{Some}(v) \rightarrow \mathsf{let}\ tail = !\, r\ \mathsf{in} \\
& \qquad \mathsf{let}\ hd = \mathbf{Some}(\mathsf{ref}(v, tail))\ \mathsf{in} \\
& \qquad \mathsf{if}\ \mathsf{CAS}(r, tail, hd) \\
& \qquad\ \mathsf{then}\ () \\
& \qquad\ \mathsf{else}\ push\ r\ mb\ v \\
pop\_no\_offer\ r\ mb = {} & \mathsf{match}\ !\, r\ \mathsf{with} \\
& \mid \mathbf{None} \rightarrow \mathbf{None} \\
& \mid \mathbf{Some}(pth) \rightarrow \mathsf{let}\ (hd, tail) = !\, pth\ \mathsf{in} \\
& \qquad \mathsf{if}\ \mathsf{CAS}(r, \mathbf{Some}(pth), tail) \\
& \qquad\ \mathsf{then}\ \mathbf{Some}(hd) \\
& \qquad\ \mathsf{else}\ pop\ r\ mb \\
pop\ r\ mb = {} & \mathsf{match}\ !\, mb\ \mathsf{with} \\
& \mid \mathbf{None} \rightarrow pop\_no\_offer\ r\ mb \\
& \mid \mathbf{Some}(off) \rightarrow \mathsf{match}\ accept\_offer\ off\ \mathsf{with} \\
& \qquad \mid \mathbf{None} \rightarrow pop\_no\_offer\ r\ mb \\
& \qquad \mid \mathbf{Some}(v) \rightarrow \mathbf{Some}(v)
\end{aligned}
$$

FIGURE 1. Stack with helping

The helper function *pop_no_offer* can itself fail (due to racing) – in that case the whole pop operation is restarted.

1.3. **Specification stack.** In order to verify the linearizability of stack with helping, we prove that it refines a coarse-grained stack, which implementation and symbolic execution rules are given in Figure 2. The internal state of the stack is a simple reference r to a list guarded by a lock lk. The reference r stores the stack value of a recursive type $\mu\alpha.\, \mathsf{unit} + \tau \times \alpha$ of lists with elements of the type τ. We have the standard constructors **cons** and **nil**.

There are standard symbolic execution rules for the right-hand side operate on an abstract predicate $\mathsf{isStack}(r_s, \vec{v})$ where $\vec{v}$ is a finite list of values. The variable r_s will by convention denote a pair of (r, lk); we use it in the specification without exposing the fact that the internal state of the stack is a pair of a location and a lock. This suggests that we do not need to stick to the exact coarse-grained stack implementation, rather we can work with any stack module that satisfies the specifications given in Figure 2.

$$
\begin{aligned}
mk\_stack_s\ () = {}& \mathsf{let}\ r = \mathsf{ref}(\mathbf{nil})\ \mathsf{in}\\
& \mathsf{let}\ lk = \mathsf{newlock}\ ()\ \mathsf{in}\\
& (\lambda().\ pop_s\ r\ lk, \lambda x.\ push_s\ r\ lk\ x)
\end{aligned}
$$

$$
\begin{aligned}
push_s\ (r, lk)\ x = {}& \mathsf{acquire}\ lk;\\
& r \leftarrow \mathbf{cons}(x, !\, r);\\
& \mathsf{release}\ lk
\end{aligned}
\qquad
\begin{aligned}
pop_s\ (r, lk) = {}& \mathsf{acquire}\ lk;\\
& \mathsf{let}\ v = \mathsf{match\ unfold}\ !\, r\ \mathsf{with}\\
& \quad \mid \mathsf{inl}(()) \to \mathbf{None}\\
& \quad \mid \mathsf{inr}(hd, tl) \to r \leftarrow tl; \mathbf{Some}(hd)\\
& \mathsf{in}\ \mathsf{release}\ lk; v
\end{aligned}
$$

$$\text{CG-PUSH-R}\quad \frac{\mathsf{isStack}(r_s, \vec{v}) \qquad (\mathsf{isStack}(r_s, x\vec{v}) \mathrel{-\!\!*} \models_{\mathcal{E}} e \precsim K[\,()\,] : \tau)}{\models_{\mathcal{E}} e \precsim K[\,push_s\ r_s x\,] : \tau}$$

$$\text{CG-POP-SUC-R}\quad \frac{\mathsf{isStack}(r_s, w\vec{v}) \qquad (\mathsf{isStack}(r_s, \vec{v}) \mathrel{-\!\!*} \models_{\mathcal{E}} e \precsim K[\,\mathbf{Some}(w)\,] : \tau)}{\models_{\mathcal{E}} e \precsim K[\,pop_s\ r_s\,] : \tau}$$

$$\text{CG-POP-FAIL-R}\quad \frac{\mathsf{isStack}(r_s, \epsilon) \qquad (\mathsf{isStack}(r_s, \epsilon) \mathrel{-\!\!*} \models_{\mathcal{E}} e \precsim K[\,\mathbf{None}\,] : \tau)}{\models_{\mathcal{E}} e \precsim K[\,pop_s\ r_s\,] : \tau}$$

FIGURE 2. Coarse-grained stack and its specification

We would like to show the following refinement:

$$\models mk\_stack \precsim mk\_stack_s : \forall \tau.\, (\mathsf{unit} \to \tau\ \mathsf{option}) \times (\tau \to \mathsf{unit}).$$

Showing this refinement boils down to coming up with an invariant $\mathsf{stackInv}(r_i, mb, r_s)$ and refinements

$$\boxed{\mathsf{stackInv}(r_i, mb, r_s)}^{\mathcal{STN}} * [\![\tau]\!](v_1, v_2) \mathrel{-\!\!*} \models push\ r_i\ mb\ v_1 \precsim push_s\ r_s\ v_2 : \mathsf{unit}$$

$$\boxed{\mathsf{stackInv}(r_i, mb, r_s)}^{\mathcal{STN}} \mathrel{-\!\!*} \models pop\ r_i\ mb \precsim pop_s\ r_s : \tau\ \mathsf{option}$$

For the later refinement we use an auxiliary conditional refinement:

$$\boxed{\mathsf{stackInv}(r_i, mb, r_s)}^{\mathcal{STN}} * (\models pop\ r_i\ mb \precsim pop_s\ r_s : \tau\ \mathsf{option}) \mathrel{-\!\!*} \models pop\_no\_offer\ r_i\ mb \precsim pop_s\ r_s : \tau\ \mathsf{option}$$

The idea is that we prove the pop refinement using Löb induction. In case the mailbox is empty or we cannot accept the offer for some other reason, we resort to calling *pop_no_offer*. However, *pop_no_offer* can fail itself, in which case it calls the *pop* function again. For that reason we have the refinement $\models pop\ r_i\ mb \precsim pop_s\ r_s : \tau\ \mathsf{option}$ as an assumption.

In the next subsections we will see how to define the stack invariant $\mathsf{stackInv}(r_i, mb, r_s)$, but before that we need to describe the specifications and the logical theory that we use for offers.

1.4. **Specification for the offers.** The specification for the offers can be constructed by considering the life cycle of an offer. The internal state of the offer is 0. A thread put ups an offer and associates with it some resource P. The thread that creates the offer also forges a token $\mathsf{offer\_token}_\gamma$; only a thread in possession

of this token can revoke an offer, transitioning it to internal state 2. This token is also used for getting the resources out of the offer: by revoking an offer a thread exchanges the token back for the resource P.

Alternatively, any thread can try to accept an offer, by changing the internal state to 1. This thread may take the resource P and put up some other resource Q in exchange.

Finally, a thread that has originally created the offer is able to see that the offer has been accepted and the resource P is taken, but a new resource Q is present. The original thread can then exchange the token $\mathsf{offer\_token}_\gamma$ for Q. This suggests the following logical description of an offer:

$$\begin{aligned}\mathsf{is\_offer}_\gamma(\ell, P, Q) \triangleq \exists c.\, \ell \mapsto_{\mathsf{i}} c * \big(&c = 0 * P \\ &\lor c = 1 * (Q \lor \mathsf{offer\_token}_\gamma) \\ &\lor c = 2 * \mathsf{offer\_token}_\gamma\big)\end{aligned}$$

where ℓ is the second component of the offer value.

Using $\mathsf{is\_offer}_\gamma(\ell, P, Q)$ and $\mathsf{offer\_token}_\gamma$ we can verify the following specifications for offer functions:

MK-OFFER-L
$$\frac{\forall \gamma\, \ell.\, (\forall P\, Q.\, P \mathrel{-\!\!*} \mathsf{is\_offer}_\gamma(\ell, P, Q)) * \mathsf{offer\_token}_\gamma \mathrel{-\!\!*} \models K[\,(v, \ell)\,] \precsim t : \tau}{\models K[\,\mathit{mk\_offer}\ v\,] \precsim t : \tau}$$

ACCEPT-OFFER-L
$$\frac{\begin{array}{c}{}^{\top}\!\Rrightarrow^{\mathcal{E}} \triangleright\, \mathsf{is\_offer}_\gamma(\ell, P, Q) * \triangleright \big((\mathsf{is\_offer}_\gamma(\ell, P, Q) \mathrel{-\!\!*} \models_{\mathcal{E}} K[\,\mathbf{None}\,] \precsim t : \tau) \\ \land (P * (Q \mathrel{-\!\!*} \mathsf{is\_offer}_\gamma(\ell, P, Q)) \mathrel{-\!\!*} \models_{\mathcal{E}} K[\,\mathbf{Some}(v)\,] \precsim t : \tau)\big)\end{array}}{\models K[\,\mathit{accept\_offer}\ (v, \ell)\,] \precsim t : \tau}$$

The specification for *revoke_offer* is more complicated and specialized for our use case: helping. It is also the only specification for which we have to open the "escape hatch" and unfold the definition of the refinement propositions. But before we divert out attention to that, we should get comfortable with the two specifications given above. Looking at MK-OFFER-L we see that we can always symbolically execute *mk_offer*, after which we get two additional premises: $(\forall P\, Q.\, P \mathrel{-\!\!*} \mathsf{is\_offer}_\gamma(\ell, P, Q))$ and $\mathsf{offer\_token}_\gamma$. Notice that this specification is written partially in an "inside out" style: instead of demanding a resource P upfront for symbolically executing *mk_offer*, the rule instead provides the user with an opportunity to turn their P into an $\mathsf{is\_offer}_\gamma(\ell, P, Q)$ at a later point at user's convenience.

The symbolic execution rule for *accept_offer* is also partially "inside out", but it is also written in a logically atomic style. The user symbolically executing *accept_offer* needs to provide $\mathsf{is\_offer}_\gamma(\ell, P, Q)$, potentially after opening an invariant. After which they need to prove two goals:

$$\mathsf{is\_offer}_\gamma(\ell, P, Q) \mathrel{-\!\!*} \models_{\mathcal{E}} K[\,\mathbf{None}\,] \precsim t : \tau$$

accounting for a failure to accept the offer, and

$$P * (Q \mathrel{-\!\!*} \mathsf{is\_offer}_\gamma(\ell, P, Q)) \mathrel{-\!\!*} \models_{\mathcal{E}} K[\,\mathbf{Some}(v)\,] \precsim t : \tau$$

accounting for a succesfull acceptance of the offer. Notice the second case the rule does not demand the user to immediately produce the resource Q (in exchange for the resources P already stored in the offer). Rather, the rule takes the resource $\mathsf{is\_offer}_\gamma(\ell, P, Q)$ as a hostage, willing to exchange it for Q.

1.5. **Verifying the refinement.** Clearly, the "inside out" rules are stronger than rules in the normal style. Let us demonstrate why do we need such rules by going through the proofs of the refinements.

First, we need to describe the stack invariant $\mathsf{stackInv}(r_i, mb, r_s)$. The invariant must support two goals: linking together the two internal representations of the stack (the coarse grained stack uses a value of type $\mu\alpha.\, \mathsf{unit} + \tau \times \alpha$, while the stack with helping uses a linked list in the memory as τ `stack`); and accounting for offers and helping.

Predicates:

$$\mathsf{offerReg}_\gamma : (\mathit{Loc} \xrightarrow{\mathsf{fin}} \mathit{Val} \times \mathit{GName} \times \mathbb{N} \times \mathit{ECtx}) \to \mathit{iProp}$$
$$\mathsf{inOfferReg}_\gamma : \mathit{Loc} \to (\mathit{Val} \times \mathit{GName} \times \mathbb{N} \times \mathit{ECtx}) \to \mathit{iProp}$$

Rules:

NEW-OFFERREG
$$\Rrightarrow \exists \gamma'.\, \mathsf{offerReg}_{\gamma'}(\emptyset)$$

OFFERREG-ALLOC
$$\frac{N(\ell) = \bot \qquad \mathsf{offerReg}_{\gamma'}(N)}{\Rrightarrow \mathsf{offerReg}_{\gamma'}(N[\ell \leftarrow (v, \gamma, j, K)]) * \mathsf{inOfferReg}_{\gamma'}(\ell, v, \gamma, j, K)}$$

OFFERREG-LOOKUP
$$\frac{\mathsf{offerReg}_{\gamma'}(N) * \mathsf{inOfferReg}_{\gamma'}(\ell, v, \gamma, j, K)}{\mathsf{offerReg}_{\gamma'}(N) * N(\ell) = (v, \gamma, j, K)}$$

INOFFERREG-PERSISTENT
$$\frac{\mathsf{inOfferReg}_{\gamma'}(\ell, v, \gamma, j, K)}{\Box\, \mathsf{inOfferReg}_{\gamma'}(\ell, v, \gamma, j, K)}$$

Defined predicates:

$$\mathsf{offerInv}(N) \triangleq \mathop{\Large *}_{\ell \mapsto (v,\gamma,j,K) \in N} \mathsf{is\_offer}_\gamma(\ell, j \Rrightarrow K[\, \mathit{push}_s\ r_s\ v\,], j \Rrightarrow K[\,()\,])$$

FIGURE 3. Offer registry ghost theory.

We have already seen the predicate $\mathsf{isStack}(r_s, \vec{v})$ that describes the contents of coarse-grained stack. In order to link it with the fine-grained representation of stack with helping, we first recursively define the predicate $\mathsf{is\_stack}_i(r_i, \vec{w})$, which describes the contents of the stack r_i in the heap; then we link the two contents together with a predicate stacklink:

$$\begin{aligned}
\mathsf{is_stack}_i(r_i, \vec{w}) &= \exists h.\, r_i \mapsto_{\mathsf{i}} h * \mathsf{is_stack}'_i(h, \vec{w}) \\
\mathsf{is_stack}'_i(\mathbf{None}, \epsilon) &= \mathsf{True} \\
\mathsf{is_stack}'_i(\mathbf{Some}(\ell), w\vec{w'}) &= \exists t\, q.\, \ell \mapsto_{\mathsf{i}} (w, t) * \mathsf{is_stack}'_i(t, \vec{w'}) \\
\mathsf{is_stack}'_i(x, y) &= \mathsf{False} \text{ otherwise}
\end{aligned}$$

$$\begin{aligned}
\mathsf{stacklink}(r_i, r_s) = \exists \vec{v}\, \vec{w}.\, \mathsf{is_stack}_i(r_i, \vec{w}) * \mathsf{isStack}(r_s, \vec{v}) * \\
(|\vec{v}| = |\vec{w}|) * \mathop{\Large *}_{0 \le i < |\vec{v}|} [\![\tau]\!](w_i, v_i)
\end{aligned}$$

To account for the offers and helping we use an *offer registry*: a map that associates each offer location ℓ to a value v associated with the offer, as well as the id and the continuation of a thread that potentially emulates the helping operation. The offer registry ghost theory keeps track of this map, ensuring that it is growing monotonically. The predicates and rules for the registry are displayed in Figure 3.

For an offer registry N we have a corresponding invariant $\mathsf{offerInv}(N)$, which actually accumulates the $\mathsf{is\_offer}_\gamma(-,-,-)$ resources. The idea is that whenever a thread accepts an offer that is in the registry it gets the resource $j \Rrightarrow K[\, \mathit{push}_s\ r_s\ v\,]$ (as per the symbolic execution rule for the offers). Then it is the job of this thread to perform helping by symbolically executing $\mathit{push}_s\ r_s\ v$ in the ghost thread pool, leaving the finished thread $j \Rrightarrow K[\,()\,]$ in the offer. Conversely, a thread that puts up an offer also offers a corresponding $j \Rrightarrow K[\, \mathit{push}_s\ r_s\ v\,]$ that can be executed by the work-stealing thread. Let us see how this idea is used in the proof of the pop refinement.

First, we write down the full invariant:

$$\mathsf{stackInv}(r_i, mb, r_s) \triangleq \exists N\, p\, v.\, \mathsf{isLock_s}(lk, \mathbf{false}) * r_i \mapsto_{\mathsf{i}} p * r_s \mapsto_{\mathsf{s}} v * \mathsf{stacklink}(p, v) * \mathsf{offerReg}_{\gamma'}(N) * \mathsf{offerInv}(N)$$
$$\big(mb \mapsto_{\mathsf{i}} \mathbf{None} \lor (\exists \ell\, v_1\, v_2\, \gamma\, j\, K.\, mb \mapsto_{\mathsf{i}} \mathbf{Some}(v_1, \ell) * [\![\tau]\!](v_1, v_2) * N(\ell) = (v_2, \gamma, j, K))\big)$$

The pop refinement that we want to prove is the following:

$$\boxed{\mathsf{stackInv}(r_i, mb, r_s)}^{\mathcal{STN}} \twoheadrightarrow \models pop\ r_i\ mb \precsim pop_s\ r_s\ lk : \tau\ \mathsf{option} \tag{1}$$

Proposition 1.1. *Refinement from Equation (1) holds.*

Proof. We proceed by symbolic execution; we open the invariant and consider two cases: whether the mailbox contains an offer or not. If the mailbox is empty, then we proceed by proving the *pop_no_offer* refinement. Otherwise the mailbox contains an offer (ℓ, v_1); furthermore we know that $N(\ell) = (v_2, \gamma, j, K)$ satisfying $[\![\tau]\!](v_1, v_2)$. From the invariant we also have $\mathsf{offerReg}_{\gamma'}(N)$;; in combination we can obtain a persistent resource $\mathsf{inOfferReg}_{\gamma'}(\ell, v_2, \gamma, j, K)$. We can successfully close the invariant and keep $\mathsf{inOfferReg}_{\gamma'}(\ell, v_2, \gamma, j, K)$ for later.

We continue with symbolic execution until we read the *accept_offer* (ℓ, v_1) sub expression. Here we apply the ACCEPT-OFFER-L rule. Once again, we open the stack invariant, which this time gives us $\mathsf{offerReg}_{\gamma'}(N')$ for some N'. However, we still have a predicate $\mathsf{inOfferReg}_{\gamma'}(\ell, v_2, \gamma, j, K)$, which proves that N' still contains the offer ℓ. Then, from the offer registry invariant, we get $\mathsf{is\_offer}_\gamma(\ell, j \Rightarrow K[\,push_s\ r_s\ v_2\,], j \Rightarrow K[\,()\,])$ which is the premise that we need for ACCEPT-OFFER-L. We are left with two goals, which are essentially[1]:

$$\mathsf{is\_offer}_\gamma(\ell, j \Rightarrow K[\,push_s\ r_s\ v_2\,], j \Rightarrow K[\,()\,]) \twoheadrightarrow \models_{\top \setminus \mathcal{STN}} pop\_no\_offer\ r_i\ mb \precsim pop_s\ r_s\ lk : \tau\ \mathsf{option}$$

and

$$j \Rightarrow K[\,push_s\ r_s\ v_2\,] * (j \Rightarrow K[\,()\,] \twoheadrightarrow \mathsf{is\_offer}_\gamma(\ell, j \Rightarrow K[\,push_s\ r_s\ v_2\,], j \Rightarrow K[\,()\,])) * [\![\tau]\!](v_1, v_2) \twoheadrightarrow \models_{\top \setminus \mathcal{STN}} \mathbf{Some}(v_1) \precsim pop_s\ r_s\ lk : \tau\ \mathsf{option}$$

The first goal is solved by closing the invariant (with the $\mathsf{is\_offer}$ resource), and proving the *pop_no_offer* refinement as standard. For the second refinement, if we want to close the invariant we need to obtain the $\mathsf{is\_offer}$ resource; for that we have to symbolically execute the $push_s\ r_s\ v_2$ operation in the ghost thread pool j. The additional benefit of doing that is that v_2 ends up on top of the stack r_s. That means that we can later symbolically execute the right-hand side of the refinement and pop v_2 right back into the right-hand side of the refinement:

$$\models_{\top \setminus \mathcal{STN}} \mathbf{Some}(v_1) \precsim \mathbf{Some}(v_2) : \tau\ \mathsf{option}.$$

Closing the invariant will complete the refinement proof. □

In order to carry out such a proof in Coq it is crucial to be able to symbolically execute expression in the ghost thread pool, which we can do by using the Coq tactics.

[1] Modulo pure reductions.

Push refinement and specification for revoke_offer. Verifying the refinement of the push operations is going to be harder. When, during the execution of *push*, we create a new offer and store it in the mailbox mb, we also have to provide a corresponding resources for the offer registry. That is, we will have to take out the right-hand side out of the refinement: $j \Rightarrow K[\,\mathit{push}_s\ r_s\ v\,]$. One approach to verifying the push refinement would be to just unfold the definition of the refinement proposition at the point where we assign $\mathbf{Some}(\mathit{off})$ to the mailbox mb, and proceed by reasoning in the WP-calculus in Iris. However, we believe that it is better to localize reasoning in WP-calculus; for that, note that despite the fact that we have to "give up" the right-hand side when we store the offer, we do get it back after *revoke_offer* is finished. Either we get the resource that we gave up ($j \Rightarrow K[\,\mathit{push}_s\ r_s\ v\,]$), or the resource that has been "helped" by some other thread ($j \Rightarrow K[\,()\,]$).

To facilitate this, we formulate the following symbolic execution rule for *revoke_offer*:

REVOKE-OFFER-L

$$\frac{\mathsf{offer\_token}_\gamma \qquad \big(\forall j\, K'.\, j \Rightarrow K'[\,e_1\,]\ {}^{\mathcal{E}}\!\Rrightarrow^{\top} \triangleright\ {}^{\top}\!\Rrightarrow^{\mathcal{E}} \triangleright \mathsf{is\_offer}_\gamma(\ell, j \Rightarrow K'[\,e_1\,], j \Rightarrow K'[\,e_2\,]) * \triangleright(\mathsf{is\_offer}_\gamma(\ell, j \Rightarrow K'[\,e_1\,], j \Rightarrow K'[\,e_2\,]) \mathrel{-\!\!*} ((\models_{\mathcal{E}} K[\,\mathbf{None}\,] \precsim e_2 : \tau) \wedge (\models_{\mathcal{E}} K[\,\mathbf{Some}(v)\,] \precsim e_1 : \tau)))\big)}{\models_{\mathcal{E}} K[\,\mathit{revoke\_offer}\ (v, \ell)\,] \precsim e_1 : \tau}$$

This rule states that we may symbolically execute *revoke_offer* in a situation when we have an invariant open (which is the situation that we end up after symbolically executing $mb \leftarrow \mathbf{Some}(\mathit{off})$), and when the right-hand side is exactly[2] e_1. In order to do that we must provide upfront an offer token $\mathsf{offer\_token}_\gamma$, and we must provide certain view shift which says the following:

- Given $j \Rightarrow K'[\,e_1\,]$ we can close the invariant, and open it again to provide $\mathsf{is\_offer}_\gamma(\ell, j \Rightarrow K'[\,e_1\,], j \Rightarrow K'[\,e_2\,])$;
- And in addition to that, we have to prove two goals, both under the assumption $\mathsf{is\_offer}_\gamma(\ell, j \Rightarrow K'[\,e_1\,], j \Rightarrow K'[\,e_2\,])$. One for when helping was successful, and some other thread has reduced e_1 to e_2, which is the new right-hand side. Another for when the helping did not happen and the right-hand side remains e_1.

The 1.5 rule is provable only through unfolding the definition of the refinement proposition and the definition of $\mathsf{is\_offer}$. However, if 1.5 is proven, then we can use it in the proof of the push refinement, thus encapsulating the place where reasoning outside ReLoC happens.

Proposition 1.2. *The following refinement holds:*

$$\boxed{\mathsf{stackInv}(r_i, mb, r_s)}^{\mathcal{STN}} * [\![\tau]\!](v_1, v_2) \mathrel{-\!\!*} \models \mathit{push}\ r_i\ mb\ v_1 \precsim \mathit{push}_s\ r_s\ \mathit{lk}\ v_2 : \mathsf{unit}$$

Proof. We start by symbolic execution. After creating a new offer (v_1, ℓ) we get $\mathsf{offer\_token}_\gamma$ and $(\forall P\, Q.\, P \mathrel{-\!\!*} \mathsf{is\_offer}_\gamma(\ell, P, Q))$. Then we symbolically execute the assignment $mb \leftarrow \mathbf{Some}(v_1, \ell)$. In order to do that we open the invariant, to get access to $mb \mapsto_{\mathsf{i}} -$. We then end up in a situation when our invariant is open, and we can symbolically execute *revoke_offer*. Note that at this point we cannot restore the invariant, because mb points to $\mathbf{Some}(v_1, \ell)$, but ℓ is not yet present in the offer registry N. We also cannot put it in the registry yet, because we do not have an $\mathsf{is\_offer}$ resource associate with it yet.

We then apply 1.5. In order to discharge the premises of that rule we have to, first and foremost, close the invariant using the resource $j \Rightarrow K'[\,\mathit{push}_s\ r_s\ \mathit{lk}\ v_2\,]$ for some j and K'. Using this resource, and the "constructor" $(\forall P\, Q.\, P \mathrel{-\!\!*} \mathsf{is\_offer}_\gamma(\ell, P, Q))$ we get $\mathsf{is\_offer}_\gamma(\ell, j \Rightarrow K'[\,\mathit{push}_s\ r_s\ \mathit{lk}\ v_2\,], j \Rightarrow K'[\,()\,])$. We can then insert ℓ into the offer registry and close the invariant.

[2]The rule can be generalized to the situation when e_1 is in under an evaluation context.

For the second step, we have to provide ${}^{\top}\Rrightarrow^{\top\setminus\mathcal{STN}} \triangleright \mathsf{is\_offer}_\gamma(\ell, j \Mapsto K'[\,push_s\ r_s\ lk\ v_2\,], j \Mapsto K'[\,()\,]) * \ldots$. That is doable: we just open the invariant and recall that ℓ was stored in the offer registry.

Lastly, we get $\mathsf{is\_offer}_\gamma(\ell, j \Mapsto K'[\,push_s\ r_s\ lk\ v_2\,], j \Mapsto K'[\,()\,])$ and we have to prove two goals.

(1) $\models_{\top\setminus\mathcal{STN}} () \precsim () : \mathsf{unit}$;
(2) $\models_{\top\setminus\mathcal{STN}} \mathsf{let}\ tail = !\,r\_i\ \mathsf{in}\ \ldots \precsim push_s\ r_s\ lk\ v_2 : \mathsf{unit}$.

In the first case we see that the offer was accepted, and we do not have anything left to do – we even have a matching right-hand side. All that remains is to close the invariant to restore the mask and verify that () refines ().

In the second case the offer was not accepted by any other thread, and we have to symbolically execute the actual push operation ourselves. The proof for that is fairly standard, and boils down to using the stacklink predicate and updating the contents of both stack at the right time. □

The full formalized proof can be found in the accompanying Coq sources.

Finally, Proposition 1.1 and Proposition 1.2 are used to prove the final refinement theorem:

$$\models mk\_stack\ () \precsim mk\_stack_s\ () : (\mathsf{unit} \to \tau\ \mathsf{option}) \times (\tau \to \mathsf{unit}).$$

As we have seen in this section, for data structures with helping we cannot carry out complete proofs inside ReLoC, and may have to resort to breaking the abstraction in order to complete the proof. However, much like semantic typing can be seamlessly combined with syntactic typing, proofs of propositions that require breaking the ReLoC abstractions can be used as parts of proofs of larger propositions in a seamless manner. It is only required that we state the intermediate propositions carefully enough to include only the refinement propositions in the conclusion and the premises.

References

[BB20] Lars Birkedal and Aleš Bizjak. Lecture notes on iris: Higher-order concurrent separation logic. https://iris-project.org/tutorial-material.html, 2020.

[HSY04] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *SPAA*, pages 206–215, 2004.

Radboud University
Email address: dfrumin@cs.ru.nl

Delft University of Technology
Email address: mail@robbertkrebbers.nl

Aarhus University
Email address: birkedal@cs.au.dk