Contextual Refinement of the Michael-Scott Queue (Proof Pearl)

Simon Friis Vindum
vindum@cs.au.dk
Aarhus University
Aarhus, Denmark

Lars Birkedal
birkedal@cs.au.dk
Aarhus University
Aarhus, Denmark

Abstract

The Michael-Scott queue (MS-queue) is a concurrent non-blocking queue. In an earlier pen-and-paper proof it was shown that a simplified variant of the MS-queue contextually refines a coarse-grained queue. Here we use the Iris and ReLoC logics to show, for the first time, that the original MS-queue contextually refines a coarse-grained queue. We make crucial use of the recently introduced prophecy variables of Iris and ReLoC. Our proof uses a fairly simple invariant that relies on encoding which nodes in the MS-queue can reach other nodes. To further simplify the proof, we extend separation logic with a generally applicable *persistent points-to predicate* for representing immutable pointers. This relies on a generalization of the well-known algebra of fractional permissions into one of *discardable* fractional permissions. We define the persistent points-to predicate entirely inside the base logic of Iris (thus getting soundness "for free").

We use the same approach to prove refinement for a variant of the MS-queue resembling the one used in the `java.util.concurrent` library.

We have mechanized our proofs in Coq using the formalizations of ReLoC and Iris in Coq.

CCS Concepts: • **Theory of computation** → **Separation logic**; **Concurrent algorithms**.

Keywords: separation logic, concurrency, Iris, Coq

ACM Reference Format:
Simon Friis Vindum and Lars Birkedal. 2021. Contextual Refinement of the Michael-Scott Queue (Proof Pearl). In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '21), January 18–19, 2021, Virtual, Denmark*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3437992.3439930

CPP '21, January 18–19, 2021, Virtual, Denmark

ACM ISBN 978-1-4503-8299-1/21/01…$15.00
https://doi.org/10.1145/3437992.3439930

1 Introduction

The Michael-Scott queue (MS-queue) is a fast and practical fine-grained concurrent queue [14]. We prove that the MS-queue is a *contextual refinement* of a coarse-grained concurrent queue. The coarse-grained queue, shown in Figure 1, is implemented as a reference to a functional list and uses a lock to sequentialize concurrent accesses to the queue. We thus prove that in any program we may replace uses of the coarse-grained, but obviously correct, concurrent queue with the faster, but more intricate, MS-queue, without changing the observable behaviour of the program. We recall that, formally, an expression e contextually refines another expression e', denoted $\Delta; \Gamma \vdash e \precsim_{ctx} e' : \tau$, if for all contexts K, of ground type, whenever $K[e]$ terminates with a value there exists an execution of $K[e']$ that terminates with the same value. One should think of e as the *implementation* (in our case the MS-queue), e' as the *specification* (in our case the coarse-grained queue), and K as a *client* of a queue implementation.

Note that the contextual refinement implies that the internal states of the two queues are *encapsulated* and hidden from clients who could otherwise tell the difference between the two implementations. Contextual refinement is also related to *linearizability*, a popular correctness criterion considered for concurrent data structures. Linearizability has mostly been considered for first-order programming languages (without higher-order functions and abstract types). For a particular first-order language and under strong assumptions, Filipovic *et al.* [7] showed that linearizability and contextual refinement coincide. Recently, Murawski and Tzevelekos [15] proposed a notion of linearizability for a programming language with higher-order functions, and they also proved that their notion of linearizability is *sound*, that is, that it implies contextual refinement. To the best of our knowledge, no sound notion of linearizability has been developed for the very rich programming language we consider (with higher types, abstract types, general references, and fork-based concurrency), so instead of using linearizability, we follow the approach of Turon et. al., and show contextual refinement directly [18].

Turon et. al. showed how the proof technique of *logical relations* can be used to prove contextual refinement of fine-grained concurrent data structures [18]. They also gave pen-and-paper proofs of contextual refinement for a simplified

$$
\begin{array}{l}
\mathsf{dequeue}_{\mathsf{CG}}\ \mathsf{lock}\ \mathsf{list}\ ()\triangleq\\
\quad\mathbf{sync}\ (\mathsf{lock})\ \{\\
\quad\quad\mathbf{match}\ \ !\mathsf{list}\ \mathbf{with}\\
\quad\quad\quad\mathsf{nil}\Rightarrow\mathsf{none}\\
\quad\quad\quad x::xs\Rightarrow\mathsf{list}\leftarrow xs;\ \mathsf{some}\ x\ \}\\
\mathsf{enqueue}_{\mathsf{CG}}\ \mathsf{lock}\ \mathsf{list}\ x\triangleq\mathbf{sync}\ (\mathsf{lock})\ \{\ \mathsf{list}\leftarrow(!\mathsf{list}+\!\!+\ [x])\ \}\\
\mathsf{queue}_{\mathsf{CG}}\triangleq\Lambda.\\
\quad\mathbf{let}\ \mathsf{lock}=\mathsf{newlock}\ ()\\
\quad\quad\ \ \mathsf{list}=\mathbf{ref}\ \mathsf{nil}\ \mathbf{in}\\
\quad(\lambda\_.\ \mathsf{dequeue}_{\mathsf{CG}}\ \mathsf{lock}\ \mathsf{list}\ (),\lambda x.\ \mathsf{enqueue}_{\mathsf{CG}}\ \mathsf{lock}\ \mathsf{list}\ x)
\end{array}
$$

Figure 1. The coarse-grained queue.

variant of the MS-queue. Here we present a mechanized proof of contextual refinement for the original MS-queue. This is more challenging, since proving refinement for it requires, among other things, the use of prophecy variables. The implementation of the MS-queue for which we prove refinement is *faithful* to the original, in the sense that we do not simplify or change it.

To carry out the proof we use ReLoC [8], a logic for reasoning about contextual refinement defined on top of Iris, a state-of-the-art higher-order concurrent separation logic framework [9]. Our mechanization uses the Coq implementations of ReLoC and Iris and the proof mode for Iris [11, 12].

A key insight in our proof is to use a notion of *reachability* as a unifying concept that concisely captures both the roles of the nodes in the MS-queue, the protocol for how the queue may be modified, and the invariants that the queue maintain. This is arguably simpler than the approach used in [18].

Like many data structures, the MS-queue contains locations that are never mutated after a certain point. To further simplify our proof we thus extend separation logic, in particular Iris, with better support for reasoning about locations that never change, by representing them as *immutable* pointers in the logic. To explain what this means at a high level, recall the points-to predicate $\ell \hookrightarrow v$, which has been present in separation logic since its inception for reasoning about shared *mutable* state [16]. The points-to predicate denotes ownership over location ℓ and the knowledge that ℓ points to the value v. It has been generalized to the *fractional* points-to predicate $\ell \hookrightarrow^q v$ where one can own a fraction, $q \in (0,1] \cap \mathbb{Q}$, of a points-to predicate [2, 3]. Changing a pointer is only possible when $q = 1$, whereas reading a location is possible with any fraction. This makes it possible to split access to a location and later reassemble it for *further mutation*. One can existentially quantify over the fraction ($\exists q. \ell \hookrightarrow^q v$) which makes it impossible to reassemble the entire fraction. This predicate, however, is only *duplicable* whereas we seek a predicate that is *persistent*—a strictly stronger notion [1]. Hence neither of these existing points-to predicates gives a satisfying way to reason about locations that arrive at a final value, after which they never change. To support reasoning about such locations, we generalize the points-to predicate further and introduce a *persistent* points-to predicate, $\ell \hookrightarrow^\square v$. In contrast to the beforementioned points-to predicates, our new persistent points-to predicate does not represent ownership over a resource; it only denotes the *knowledge* that ℓ always points to v. Since this predicate is persistent in the Iris-technical sense, it satisfies additional properties in comparison to the standard (fractional) points-to predicate and reasoning about immutable locations therefore becomes simpler when this predicate is used. We show that one can obtain a persistent points-to predicate by generalizing the notion of fractional permissions to one that allows *discarding* a fraction. One can then discard a fraction of the fractional points-to predicate and obtain a persistent points-to predicate; intuitively this makes sense since changing a location requires the entire fraction of the points-to predicate.

In summary, we make the following contributions:

- We show how the invariants maintained by the MS-queue can be expressed in a simple and unifying way by a notion of *reachability*.
- We show that a faithful implementation of the original MS-queue contextually refines a coarse-grained queue.
- We extend separation logic (Iris and ReLoC in particular) with a persistent points-to predicate and demonstrate how it simplifies reasoning about the MS-queue.
- We show how the persistent points-to predicate and its associated proof rules can be defined and proven entirely inside the Iris base logic.
- To define the persistent points-to predicate we construct two novel resource algebras. The resource algebra of *discardable fractions*, which generalizes the well-known notion of fractions in separation logic, and the authoritative resource algebra with projections.
- Based on our formal proof, we discover that the use of *consistent snapshots* in the MS-queue is not necessary for the correctness of the algorithm in a garbage collected language.
- Finally, we use the same approach based on reachability to prove refinement for a variant of the MS-queue resembling the one used in the `java.util.concurrent` library.

All our results are formalized in Coq and we have extended the Coq implementation of Iris and ReLoC to support the persistent points-to predicate [20].

Outline. We explain the fine-grained MS-queue algorithm and its implementation in Section 2 and then proceed to describe the structure of a refinement proof in ReLoC in Section 3, where we also present the coarse-grained queue that serves as a specification. The persistent points-to predicate and its proof rules are introduced in Section 4. Here we also further motivate why we seek a points-to predicate that is persistent and not merely duplicable. In Section 5 we

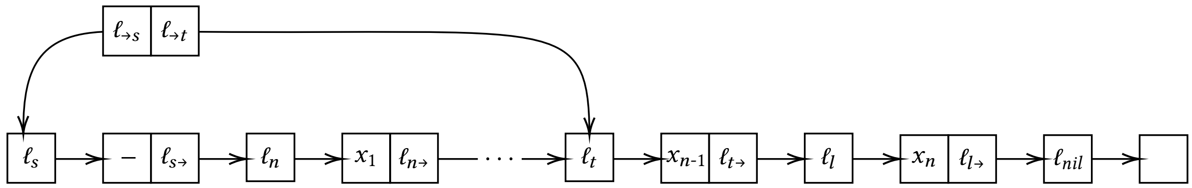

Figure 2. The MS-queue consists of a singly linked list. Here the tail pointer is lagging as it points to the second to last node.

detail the key ideas of the refinement proof and the invariant used. In Section 6 we present the actual refinement proof. In Section 7 we observe that the so-called consistent snapshots used in the MS-queue can be omitted without compromising the correctness of the algorithm, and in Section 8 we quickly comment on how we have used the same proof technique to prove refinement for a variant of the MS-queue. Finally, in Section 9 we detail how the persistent points-to predicate and its properties are actually defined and proved in the Iris base logic, by introducing two novel resource algebras. While we do recall the notion of a resource algebra, some familiarity with the Iris notion of resource algebras is probably needed to understand the details of (only) this section. We end by discussing related work in Section 10.

$$
\begin{aligned}
\tau &::= \alpha \mid 1 \mid \mathsf{bool} \mid \mathsf{int} \mid \tau \times \tau \mid \tau + \tau \mid \tau \to \tau \\
&\mid\ \forall \alpha.\tau \mid \exists \alpha.\tau \mid \mu\alpha.\tau \mid \mathsf{ref}\ \tau \\
v &::= i \in \mathbb{Z} \mid \ell \in \mathit{Loc} \mid \mathbf{true} \mid \mathbf{false} \mid (v, v) \mid \mathbf{inj}_1\, v \mid \mathbf{inj}_2\, v \\
&\mid\ \mathbf{rec}\, f(x) = e \mid \Lambda.e \mid \mathbf{pack}\ v \mid \mathbf{fold}\ v \\
e &::= x \mid v \mid \mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e \mid (e, e) \mid \pi_1\ e \mid \pi_2\ e \mid \mathbf{inj}_1\ e \mid \mathbf{inj}_2\ e \\
&\mid\ \mathbf{match}\ e\ \mathbf{with}\ \mathbf{inj}_1\, x \Rightarrow e \mid \mathbf{inj}_2\, x \Rightarrow e \mid e\ e \mid e\langle\rangle \\
&\mid\ \mathbf{pack}\ e \mid \mathbf{unpack}\ e\ \mathbf{in}\ \ x.e \mid \mathbf{fold}\ e \mid \mathbf{unfold}\ e \\
&\mid\ \mathbf{ref}(e) \mid\ !e \mid e \leftarrow e \mid \mathbf{CAS}(e, e, e) \mid \mathbf{fork}\ \{e\} \mid \ldots
\end{aligned}
$$

Syntactic sugar

$$\mathsf{Option}\ \tau \triangleq 1 + \tau \quad \mathsf{none} \triangleq \mathbf{inj}_1\, 1 \quad \mathsf{some}\ v \triangleq \mathbf{inj}_2\, v$$

$$\lambda x.\, e \triangleq \mathbf{rec}\ \_\ x = e \quad \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \triangleq (\lambda x.\, e_2)\ e_2$$

Figure 3. Syntax of the types and terms of HeapLang.

2 The MS-Queue

As depicted in Figure 2, the MS-queue consists of a singly linked list that contains the values ($x_1, \ldots, x_n$ in the figure) in the queue. The first node (ℓ_s) is called the *sentinel* and its content is not a value in the queue. The queue maintains two pointers, the *sentinel pointer* ($\ell_{\to s}$), which points to the sentinel, and the *tail pointer* ($\ell_{\to t}$), which points to the *tail* (ℓ_t). The tail is either equal to the *last node* (ℓ_l) or the second to last node. In the latter case, we say that the tail pointer is *lagging behind*. Note that $\ell_t = \ell_l$ when the tail pointer is *not* lagging behind.

We adopt the following naming convention: If ℓ_n is a location representing a node, then a location pointing *into* that node is denoted $\ell_{\to n}$ and the location pointing *out* from that node to the next node is denoted $\ell_{n\to}$. If ℓ_n is a node and ℓ_m its successor, then the pointer between the nodes can be denoted both $\ell_{n\to}$ or $\ell_{\to m}$ depending on the circumstances.

The implementation of the MS-queue is shown in Figure 7. It is written in HeapLang, a language included in the mechanization of Iris and which ReLoC extends with a type system to facilitate refinement proofs. The syntax of the language is presented in Figure 3, it is a λ-calculus with impredicative polymorphism, iso-recursive types, higher-order store, and thread-based concurrency. The language and its type system are standard; further details can be found in [8].

We have kept our implementation as faithful as possible to the original implementation. In order to emphasize this, we have annotated the code with line numbers in direct correspondence with the line numbers in Michael and Scott's original code [14]. All differences are minor and stem from inherent differences between HeapLang and the C-like language used in the original.

Initialization. The $\mathsf{queue}_{\mathsf{MS}}$ function is the constructor for the queue and the entry point to the implementation. It uses a type abstraction, Λ, such that the queue is generic in the type of elements that it stores. This lambda also serves to ensure that the internal state of the queue is encapsulated in a closure. The initialization allocates an initial node, a sentinel pointer, and a tail pointer. The latter two points to the initial node. A newly constructed queue is illustrated in Figure 4.

A node is a pointer to either none or some of a pair of a value and a pointer to the next node. The pointer serves to make nodes comparable by pointer equality such that pointers to nodes can be changed with **CAS**.

Since there is no value to put in the initial sentinel, which $\mathsf{queue}_{\mathsf{MS}}$ must construct, none is used. All other nodes contain an actual value v and hence contains some v. Thus we often need to get the value of an Option which is known to be a some. This is the purpose of the getValue function.

Dequeue. Dequeue reads the sentinel pointer and then the pointer to the sentinel's successor. If no successor exists the queue is empty and none is returned. If a succeeding node is found, dequeue attempts to change the sentinel pointer to the succeeding node with **CAS**. If the **CAS** is successful, the value in the new sentinel is returned. If the **CAS** is unsuccessful the operation is restarted. Figure 5 shows how

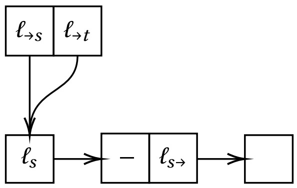

Figure 4. A newly constructed queue.

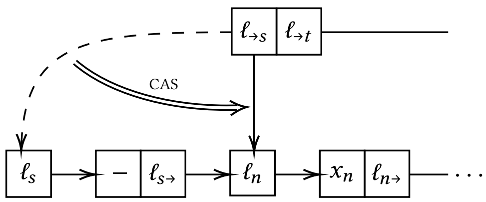

Figure 5. dequeue on the MS-queue.

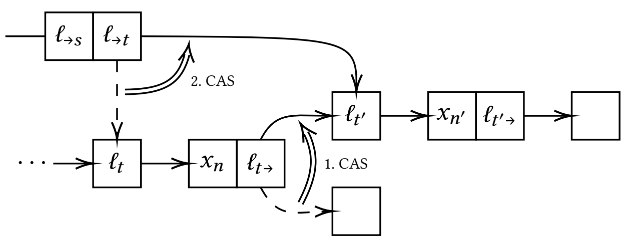

Figure 6. enqueue on the MS-queue.

successfully dequeuing an element from a non-empty queue swings the sentinel pointer forward.

The implementation contains prophecy annotations on line D4b and D5. These do not affect the execution of the program and can be ignored for now.

Enqueue. Enqueue constructs a new node with the value that is to be enqueued. It then reads the tail pointer and obtains a node that may be the last. To determine if it is, enqueue checks whether or not the node has a successor. If a successor exists the tail pointer is lagging behind, and enqueue attempts to move the tail pointer forward with a **CAS** after which it restarts. If no successor exists then the node is currently the last. By means of a **CAS** enqueue then attempts to change the outgoing pointer of the node such that it points to the new node. If the **CAS** is successful, the tail pointer now lags behind, and enqueue attempts to advance the tail pointer to the new node. If, on the other hand, the **CAS** is unsuccessful, the operation restarts, and the tail pointer is read anew. Figure 6 illustrates how a successful enqueue inserts a new node and then swings the tail pointer forward.

Highlights. We highlight a few aspects of the MS-queue that are of particular interest in terms of the verification.

On D6 the sentinel and tail are compared to each other. This is a rather indirect way of checking whether or not the queue is empty. If they are equal the queue is either empty or the tail pointer lags behind. Otherwise, the **else** branch on line D13 assumes that the queue is guaranteed to be non-empty. In our proof, we must formalize why this assumption is correct.

On line D5, a so-called *consistent snapshot* is performed: the value of toSent read on line D2 is compared to a newly read value of toSent. This ensures that toSent has not changed in the meantime and is intended to ensure that the values of tail and next are consistent. Similarly, enqueue performs a consistent snapshot on line E7.

Line D7 checks whether the next node is none or not. If it is not, then the tail pointer is lagging behind because an unfinished enqueue operation has not yet updated it. Dequeue then attempts to update the tail pointer on D10. Likewise, on E13 enqueue also detects a lagging tail and attempts to update it. These are instances of *helping*, a pattern where the execution of one operation helps another.

As we will see, a contextual refinement proof for a fine-grained concurrent data-structure involves finding its *linearization points*. It is fairly clear that enqueue's linearization point is the **CAS** on E9 and that dequeue has a linearization point on line D13. What is less obvious is that when dequeue finds the queue empty and returns none on D8, its linearization point is at the load on D4c. However, line D4c is only a linearization point if next points to none *and* if the consistent snapshot on the *next line* succeeds. Because of this, it was conjectured by Morten Krogh-Jespersen[1] that one would need some kind of prophecy variables to reason about this; and indeed, in our proof, to know whether or not the check on the next line succeeds we use the recently introduced *prophecy variables* of Iris and ReLoC.

3 Structure of a Refinement Proof

In this section, we describe how to carry out a refinement proof of a fine-grained concurrent data-structure such as the MS-queue using ReLoC. We first consider the ingredients that such a proof consists of.

Persistently modality. Iris has a persistently modality $\Box$ and $\Box\, P$ means that P *always holds*. A proposition P is per definition *persistent* if $P \vdash \Box\, P$, *i.e.,* if one from P alone can show that P always holds. Therefore persistent propositions represent *knowledge*. Propositions that are not persistent are called *ephemeral*—they represent ownership over resources. To show a goal of the form $\Box\, P$ one can only use persistent assumptions (PERSISTENT-$\Box$ in Figure 8). The intuition being that to show something always holds one can only depend on other facts that always hold.

Specification. In a proof of refinement, the *specification* should be a simple implementation of the same *interface* that

[1] When he attempted to verify the MS-queue in 2014 using the iCap logic, a precursor to Iris. Private communication.

```
         getValue x ≜ match x with none ⇒ () () | some v ⇒ v
1:       queue_MS ≜ Λ.
2:         let node = ref (some(none, (ref (ref none))))
3:             tail = ref node
4:             sent = ref node
5:         in (dequeue_MS sent tail, enqueue_MS tail)
D1:      dequeue_MS toSent toTail ≜ rec loop () =
D2:        let sent = !toSent
D3:            tail = !toTail
D4a:           toNext = π2 (getValue !sent)
D4b:           p = NewProph
D4c:           next = !toNext in
D5:        if sent = Resolve(!toSent, p, ()) then
D6:          if sent = tail then
D7:            match !next with
D8:              none ⇒ none
D10:             some _ ⇒ CAS toTail tail next; loop ()
D11:         else
D13:           if CAS toSent sent next
D14:           then some (getValue(π1(getValue !next)))
D15:           else loop ()
D16:       else loop ()
         enqueue_MS toTail x ≜
E1-E3:     let node = ref (some (some x, ref (ref none))) in
E4:        (rec loop() =
E5:          let tail = !toTail
E6a:             toNext = π2 (getValue !tail)
E6b:             next = !toNext in
E7:          if tail = !toTail then
E8:            match !next with
E9:              none ⇒ if CAS toNext next node
E17:                    then CAS toTail tail node; ()
E11:                    else loop ()
E13:             some _ ⇒ CAS toTail tail next; loop ()
E14:         else loop ()) ()
```

Figure 7. Implementation of the MS-queue in HeapLang.

the *implementation* is intended to implement. As mentioned in the Introduction, our specification is a coarse-grained concurrent queue, implemented using a pointer to a functional list and where the operations are guarded by a lock, which is included in ReLoC. The official definition of the coarse-grained queue is given in Figure 9; the version shown in the Introduction used a modicum of syntactic sugar.

Refinement judgment. To prove a contextual refinement ReLoC offers a *refinement judgment* $\models e_1 \precsim e_2 : \tau$ which denotes that e_1 refines e_2 at the type τ. The ReLoC soundness theorem states that if such a judgment holds inside the logic, then the corresponding contextual refinement holds in the surrounding meta-logic. ReLoC provides high-level rules for working with these refinement judgments that result in simpler proofs than other approaches (*e.g.*, directly using logical relations). The *structural rules* apply when each side of the refinement is of the same syntactic form—it then suffices to show refinement of the sub-expressions that constitute the constructions. One such rule is REL-PAIR, which states that to show that two pairs are related it suffices to show that they are pair-wise related. Note that to show that two functions are related, using REL-REC, one must do so persistently, that is, without relying on any ephemeral resources. This is because a context could call a function an arbitrary number of times, and thus the functions must *always* be related at any point in the future.

When the two sides of the refinement are not of the same syntactic form, one must use *symbolic execution rules* to step either side forward. Note that the i and s in the points-to predicates denote if they are for the implementation or the specification.

Invariants. As mentioned, to show that two functions are related one can only use persistent propositions. Non-persistent propositions can be made persistent by establishing an *invariant* using the rule INV-ALLOC. The proposition $\boxed{P}^{\iota}$ denotes knowledge of an invariant with the name ι and is persistent even if P is not. During a refinement proof, one can *open* an invariant around a single atomic expression e on the left-hand side. The contents of the invariant can be used to symbolically execute e, but, afterward it is an obligation to close the invariant by showing that it still holds. Crucially this restriction does not apply to the right-hand side, here it is allowed to take several steps of symbolic execution with an invariant open. The way the above restrictions are enforced is rather technical, so we omit the details, but note that the modality $\models\!\!\!\Rrightarrow$ is used to denote when invariants can be opened.

Linearization points. During a refinement proof, one must maintain a link between the state of the implementation and the specification such that upon termination one can show that the two values are related. For a fine-grained concurrent data-structure, such as the MS-queue, operations "take effect" at specific points, namely the linearization points. At these points, the specification should be symbolically executed from start to end; this is possible even while an invariant is open per the above. To this end we use the rules for the coarse-grained queue shown in Figure 10; these are easy to prove using the lock specification that ReLoC includes, and our definition of the *representation predicate* I_{CG} for the coarse-grained queue, also shown in the figure. The representation predicate states that the physical state of the coarse-grained queue (the pointer to a list and the lock) corresponds to a logic-level sequence.

Prophecy variables. For the MS-queue in particular we also need *prophecy variables*. These are a recent addition to Iris and ReLoC [8, 10]. Recall how the load at D4c may be a linearization point depending on the result of the load on the next line, D5. Hence, when we symbolically execute the load

$$\textsc{$\square$-sep-and}\ \frac{\square(P \land Q)}{\square(P * Q)} \qquad \textsc{$\square$-exists}\ \frac{\exists x.\, \square P}{\square\, \exists x.\, P} \qquad \textsc{persistent-$\square$}\ \frac{P \text{ persistent} \qquad P \vdash Q}{P \vdash \square Q} \qquad \textsc{inv-alloc}\ \frac{P}{{}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}} \boxed{P}^{\iota}} \qquad \textsc{löb}\ \frac{Q \land \triangleright P \vdash P}{Q \vdash P}$$

Structural rules

$$\textsc{rel-return}\ \frac{[\![\tau]\!]_\Delta(v_1, v_2)}{\Delta \models v_1 \precsim v_2 : \tau} \qquad \textsc{rel-tlam}\ \frac{\forall R : Val \times Val \to \mathrm{Prop}.\ \square\left([\alpha := R], \Delta \models e_1 \precsim e_2 : \tau\right)}{\Delta \models \Lambda.e_1 \precsim \Lambda.e_2 : \forall \alpha.\tau} \qquad \textsc{rel-pair}\ \frac{\Delta \models e_1 \precsim e_2 : \tau \qquad \Delta \models e_1' \precsim e_2' : \sigma}{\Delta \models (e_1, e_1') \precsim (e_2, e_2') : \tau \times \sigma}$$

$$\textsc{rel-rec}\ \frac{\square\left(\forall v_1, v_2.\, [\![\tau]\!]_\Delta(v_1, v_2) \twoheadrightarrow \Delta \models (\mathbf{rec}\, f_1(x_1) = e_1)\, v_1 \precsim (\mathbf{rec}\, f_2(x_2) = e_2)\, v_2 : \sigma\right)}{\Delta \models (\mathbf{rec}\, f_1(x_1) = e_1) \precsim (\mathbf{rec}\, f_2(x_2) = e_2) : \tau \to \sigma}$$

Symbolic execution rules

$$\textsc{rel-pure-r}\ \frac{e_2 \overset{\text{pure}}{\rightsquigarrow} e_2' \qquad \Delta \models_{\mathcal{E}} e_1 \precsim K[\, e_2' \,] : \tau}{\Delta \models_{\mathcal{E}} e_1 \precsim K[\, e_2 \,] : \tau} \qquad \textsc{rel-load-r}\ \frac{\ell \hookrightarrow_{\mathsf{s}} v \qquad \ell \hookrightarrow_{\mathsf{s}} v \twoheadrightarrow \Delta \models_{\mathcal{E}} e_1 \precsim K[\, v \,] : \tau}{\Delta \models_{\mathcal{E}} e_1 \precsim K[\, !\,\ell \,] : \tau} \qquad \textsc{rel-store-r}\ \frac{\ell \hookrightarrow_{\mathsf{s}} - \qquad \ell \hookrightarrow_{\mathsf{s}} v \twoheadrightarrow \Delta \models_{\mathcal{E}} e_1 \precsim K[\, () \,] : \tau}{\Delta \models_{\mathcal{E}} e_1 \precsim K[\, \ell \leftarrow v \,] : \tau}$$

Rules for prophecy variables

$$\textsc{rel-newproph-l}\ \frac{\forall v, p.\, \mathrm{Proph}_1(p, v) \twoheadrightarrow \Delta \models K[\, p \,] \precsim e_2 : \tau}{\Delta \models K[\, \mathit{NewProph} \,] \precsim e_2 : \tau} \qquad \textsc{rel-resolveproph-l}\ \frac{\mathrm{Proph}_1(p, v) \qquad \mathrm{wp}\ e\ \{u.\, v = (u, w) \twoheadrightarrow \Delta \models_{\mathcal{E}} K[\, v \,] \precsim e_2 : \tau\}}{\Delta \models K[\, \mathit{Resolve}(e, p, w) \,] \precsim e_2 : \tau}$$

Figure 8. Selected rules from ReLoC (some are simplified for the sake of presentation).

```
dequeue′_CG list ≜
  match !list with
    none ⇒ none
    some p ⇒ list ← (π₂ p); some (π₁ p)
dequeue_CG lock list () ≜
  acquire lock; let v = dequeue′_CG list in release lock; v
enqueue′_CG ≜ rec loop x list =
  match list with
    none ⇒ some (x, none)
    some p ⇒ some (π₁ p, loop x (π₂ p))
enqueue_CG lock list x ≜
  acquire lock; list ← enqueue′_CG x !list; release lock
queue_CG ≜ Λ.
  let lock = newlock ()
      list = ref none in
  (λ_. dequeue_CG lock list (), λx. enqueue_CG lock list x)
```

Figure 9. Implementation of the coarse-grained queue.

$$\mathrm{I_{CG}}(\ell_{cg}, lk, xs) \triangleq \ell_{cg} \hookrightarrow_{\mathsf{s}} \mathrm{isList}(xs) * \mathrm{isLocked}(lk, \mathrm{False})$$
$$\mathrm{isList}([\,]) \triangleq \mathrm{none}$$
$$\mathrm{isList}(x :: xs) \triangleq \mathrm{some}\ (x, \mathrm{isList}(xs))$$

$$\textsc{dequeue$_{\text{CG}}$-nil-r}\ \frac{\mathrm{I_{CG}}(\ell_{cg}, lk, [\,]) \qquad \mathrm{I_{CG}}(\ell_{cg}, lk, [\,]) \twoheadrightarrow \Delta \models_{\mathcal{E}} e_1 \precsim K[\, \mathrm{none} \,] : \tau}{\Delta \models_{\mathcal{E}} e_1 \precsim K[\, \mathrm{dequeue_{CG}}\ lk\ \ell_{cg}\ () \,] : \tau}$$

$$\textsc{dequeue$_{\text{CG}}$-cons-r}\ \frac{\begin{array}{c}\mathrm{I_{CG}}(\ell_{cg}, lk, x :: xs) \\ \mathrm{I_{CG}}(\ell_{cg}, lk, xs) \twoheadrightarrow \Delta \models_{\mathcal{E}} e_1 \precsim K[\, \mathrm{some}\ x \,] : \tau\end{array}}{\Delta \models_{\mathcal{E}} e_1 \precsim K[\, \mathrm{dequeue_{CG}}\ lk\ \ell_{cg}\ () \,] : \tau}$$

$$\textsc{enqueue$_{\text{CG}}$-r}\ \frac{\mathrm{I_{CG}}(\ell_{cg}, lk, xs) \qquad \mathrm{I_{CG}}(\ell_{cg}, lk, xs \mathbin{+\!\!+} [x]) \twoheadrightarrow \Delta \models_{\mathcal{E}} e_1 \precsim K[\, () \,] : \tau}{\Delta \models_{\mathcal{E}} e_1 \precsim K[\, \mathrm{enqueue_{CG}}\ lk\ \ell_{cg}\ x \,] : \tau}$$

Figure 10. Right-hand side relational specification for the coarse-grained queue.

at D4c we need to know the result of a *future* expression. This is what prophecy variables make possible. They rely on code annotations, which do not affect the execution of the program but aids in reasoning. A prophecy is created with *NewProph* and per rel-newproph-l it results in a resource $\mathrm{Proph}_1(p, v)$ where p is the name of the prophecy and v is a value. Intuitively, v is equal to the value which the prophecy is eventually resolved to. A prophecy is resolved with an *atomic prophecy resolution*: *Resolve*(e, p, w). This expression behaves computationally exactly as the atomic expression e. Its rule rel-resolveproph-l requires $\mathrm{Proph}_1(p, v)$, and hence one can think of this resource as giving one the right to resolve the prophecy. It then states that v is equal to (u, w) where u is that value that e evaluates to. In our case we create a prophecy at D4b, hence at this point we get a value v that can be thought of as the result of the future expression !toSent.

Given these ingredients, the overall structure of a refinement proof is: *(a)* Decide on a specification and prove right–hand side lemmas for each operation (Figure 10 in our case).

$$\textsc{Mapsto-intro-}\square\ \frac{\ell \hookrightarrow_{\mathsf{i}}^{q} v}{\Rrightarrow \ell \hookrightarrow_{\mathsf{i}}^{\square} v} \qquad \textsc{Mapsto-agree-}\square\ \frac{\ell \hookrightarrow_{\mathsf{i}}^{\square} v \qquad \ell \hookrightarrow_{\mathsf{i}}^{\square} v'}{v = v'} \qquad \textsc{persistent}\ \frac{\ell \hookrightarrow_{\mathsf{i}}^{\square} v}{\square\, \ell \hookrightarrow_{\mathsf{i}}^{\square} v}$$

$$\textsc{Ht-load-}\square\ \frac{}{\{\ell \hookrightarrow_{\mathsf{i}}^{\square} u\}\,!\,\ell\{v.v = u\}} \qquad \textsc{rel-load-r-}\square\ \frac{\ell \hookrightarrow_{\mathsf{i}}^{\square} v \qquad \Delta \models K[\,v\,] \precsim e_2 : \tau}{\Delta \models K[\,!\,\ell\,] \precsim e_2 : \tau}$$

$$\textsc{rel-cas-l}\ \frac{{}^{\top}\!\!\Rrightarrow^{\mathcal{E}} \exists v. \begin{pmatrix} (v \neq v_1 \mathrel{-\!\!*} \\ \quad (\ell \hookrightarrow_{\mathsf{i}}^{\square} v * \Delta \models_{\mathcal{E}} K[\,\mathbf{false}\,] \precsim e_2 : \tau) \vee \\ \quad \exists q.\, (\ell \hookrightarrow_{\mathsf{i}}^{q} v * \\ \qquad (\ell \hookrightarrow_{\mathsf{i}}^{q} v \mathrel{-\!\!*} \Delta \models_{\mathcal{E}} K[\,\mathbf{false}\,] \precsim e_2 : \tau))) \wedge \\ (v = v_1 \mathrel{-\!\!*} \\ \quad (\ell \hookrightarrow_{\mathsf{i}} v * (\ell \hookrightarrow_{\mathsf{i}} v_2 \mathrel{-\!\!*} \Delta \models_{\mathcal{E}} K[\,\mathbf{true}\,] \precsim e_2 : \tau))) \end{pmatrix}}{\Delta \models K[\,\mathbf{CAS}(\ell, v_1, v_2)\,] \precsim e_2 : \tau}$$

Figure 11. Rules for the persistent points-to predicate.

(b) Define an invariant that relates the state of the specification to that of the implementation (Section 5) *(c)* Use symbolic execution rules to step through the initialization of each side. *(d)* Establish the invariant and use structural rules to get the goals to show that each operation is related. *(e)* Show that each operation is related by using the invariant; at each linearization point apply the corresponding lemma for the specification.

4 Persistent Points-to Predicate

Consider the depiction of the MS-queue in Figure 2 on page 3. All the pointers, except $\ell_{\rightarrow s}$, $\ell_{\rightarrow t}$, and $\ell_{l\rightarrow}$, are never changed, and, once $\ell_{l\rightarrow}$ is changed it is never changed again. As we will see, expressing precisely which parts of the MS-queue change, and which do not, is central to our approach. Since data-structures with locations that are or become immutable are common, it makes sense to develop a generally applicable tool for reasoning about immutable pointers. To this end, we introduce the *persistent points-to predicate*, denoted $\ell \hookrightarrow_{\mathsf{i}}^{\square} v$ as mentioned in the Introduction. In contrast to the normal points-to predicate, which allows for mutation but no sharing, the persistent points-to predicate allows for free sharing but no mutation.

The reader may wonder whether there is an already existing alternative to a new persistent points-to predicate. Perhaps $\exists q.\, \ell \hookrightarrow_{\mathsf{i}}^{q} v$? This predicate, however, is only duplicable whereas we want a points-to predicate that is persistent. This is because persistence is a strictly stronger notion and persistent propositions enjoy additional properties. The persistent modality commutes with all the logical connectives (*e.g.*, $\square$-exists) and under it conjunction and separating conjunction coincides ($\square$-sep-and). Hence persistent propositions form a sublogic with non-substructural properties. This is not the case for duplicable propositions: for instance, $\ell \hookrightarrow v$ is *not* duplicable but $\exists q. \ell \hookrightarrow^{q} v$ is. Persistent propositions are utilized to great effect in the Coq mechanization of Iris, see [12].

Maybe one could remedy this issue by wrapping the existentially quantified fractional points-to predicate in an invariant, that is, use $\boxed{\exists q. \ell \hookrightarrow^{q} v}^{\iota}$? This would result in a persistent predicate, but, we want a persistent points-to predicate that can be used as a normal points-to predicate, including being put inside invariants, and with this definition, we would be led to nested invariants. And while Iris does support nested invariants, reasoning about such would involve the later modality and, as a result, it would make the use of the persistent points-to predicates more restrictive.

Other approaches to modeling immutable locations exist, e.g., one may use a combination of invariants and additional ghost state, as done in [12], but this approach is more complex and our points-to predicate would have simplified the proofs in [12].

A selection of the rules for the persistent points-to predicate is shown in Figure 11. Since the persistent points-to predicate represents locations that never change, it is persistent (persistent). Given *any fraction* of a normal points-to predicate, one can obtain a persistent points-to predicate (Mapsto-intro-$\square$)—one can think of the fractional points-to predicate as being discarded in exchange for a persistent points-to predicate. The modality $\Rrightarrow$ is there because discarding the fraction requires updating ghost state. Persistent points-to predicates for the same location must point to the same value (Mapsto-agree-$\square$). Finally, the predicate can be used for read-only operations, such as loading a pointer (Ht-load-$\square$).

In Section 9 we show how to define the persistent points-to predicate and derive its rules entirely *within the Iris base logic*. This automatically guarantees soundness of the rules. We have additionally extended the Coq formalization of Iris and ReLoC to support the persistent points-to predicate as seamlessly as they support the normal points-to predicate. Among other things, this means that the tactics in the proof mode automatically use the persistent points-to predicate when possible.

The last rule in Figure 11, rel-cas-l, is an improved version of a corresponding rule in ReLoC [8]. It now allows using the persistent points-to predicate to show that a failed **CAS** is safe. This makes sense since it is sufficient to have read-only access to a location as long as one is not actually successful in mutating it. The other change to the rule is in the ordering of connectives. This change is subtle but makes the rule more complete. The original rule for **CAS** in ReLoC is structured as

$$\exists v.\, \ell \hookrightarrow_{\mathsf{i}} v * ((v \neq v_1 \mathrel{-\!\!*} \dots) \vee (v = v_1 \mathrel{-\!\!*} \dots))$$

whereas our rule allows one to first offer a witness v, then assume either $v = v_1$ or $v \neq v_1$, and then *use this (in)equality*

$$\textsc{abs-reach-alloc}\;\frac{\ell_n \rightsquigarrow \ell_n}{\exists \gamma_n.\, \gamma_n \Mapsto \ell_n * \ell_n \dashrightarrow \gamma_n} \qquad \textsc{abs-reach-concr}\;\frac{\ell_n \dashrightarrow \gamma_m \qquad \gamma_m \Mapsto \ell_m}{\ell_n \rightsquigarrow \ell_m * \gamma_m \Mapsto \ell_m}$$

$$\textsc{abs-reach-abs}\;\frac{\ell_n \rightsquigarrow \ell_m \qquad \gamma_m \Mapsto \ell_m}{\Rrightarrow (\ell_n \dashrightarrow \gamma_m * \gamma_m \Mapsto \ell_m)} \qquad \textsc{abs-reach-advance}\;\frac{\gamma_m \Mapsto \ell_m \qquad \ell_m \rightsquigarrow \ell_o}{\Rrightarrow (\gamma_m \Mapsto \ell_o * \ell_o \dashrightarrow \gamma_m)}$$

Figure 12. Rules for abstract reachability.

to show the points-to predicate. This turns out to be essential in the proof of refinement of enqueue.

5 Invariant for the Refinement Proof

We now present the invariant used in the refinement proof.

5.1 Reachability

A key insight of our approach is how the invariants that the MS-queue maintains can be expressed in terms of which nodes are *reachable* from other nodes. Reachability is expressed with an inductive predicate:

$$\ell_n \rightsquigarrow \ell_m \triangleq \exists \ell_{n\rightarrow}, v.\, \ell_n \hookrightarrow_{\mathsf{i}}^{\square} \mathsf{some}(v, \ell_{n\rightarrow}) * \\ (\ell_n = \ell_m \vee \exists \ell_p.\, \ell_{n\rightarrow} \hookrightarrow_{\mathsf{i}}^{\square} \ell_p * \ell_p \rightsquigarrow \ell_m)$$

It is persistent as the definition uses the persistent points-to predicate to express that the sequence of nodes is immutable.

Reachability is a preorder on nodes in the sense that for all ℓ_n and ℓ_m:

$$\ell_n \hookrightarrow_{\mathsf{i}}^{\square} \mathsf{some}\,(v, \ell_{n\rightarrow}) \mathrel{*\!\!-\!\!*} \ell_n \rightsquigarrow \ell_n \qquad \text{(reachable-reflexive)}$$

$$\ell_n \rightsquigarrow \ell_m \mathrel{-\!\!*} \ell_m \rightsquigarrow \ell_o \mathrel{-\!\!*} \ell_n \rightsquigarrow \ell_o \qquad \text{(reachable-transitive)}$$

Note, that $\ell_n \rightsquigarrow \ell_n$ is not trivial, it implies that ℓ_n is actually a node, in the sense that it points to some of a pair. More generally, $\ell_n \rightsquigarrow \ell_m$ implies that both ℓ_n and ℓ_m are nodes.

5.2 Abstract Reachability

A crucial property of the MS-queue is that the sentinel and tail pointers are only moved *forward* to succeeding nodes. Additionally, the linked list is never mutated except when new nodes are added at the very end. This implies that if a node can reach the *current* sentinel, tail, or last node then it can reach any *future* sentinel, tail, or last node.

To model this we use three ghost variables, γ_s, γ_t, and γ_l, as *abstract nodes* that give fixed names to the idea of the "current" sentinel, tail, and last node respectively. We then introduce *abstract reachability*, $\ell_n \dashrightarrow \gamma_m$, capturing that the *physical* node ℓ_n can reach the *abstract* node γ_m. To realize this intention, our invariant will *tie* the three abstract nodes to the locations that are currently the sentinel, tail, and last nodes. This is done using a predicate $\gamma_n \Mapsto \ell_m$ representing that the abstract node γ_n is currently tied to the physical node ℓ_m.

These predicates satisfy the rules given in Figure 12. The first two rules state that given $\gamma_m \Mapsto \ell_m$ one can go from $\ell_n \rightsquigarrow \ell_m$ to $\ell_n \dashrightarrow \gamma_m$, and vice versa. The last rule makes it possible to change which physical node an abstract node is tied to as long as the new node is reachable from the current node.

For the reader familiar with Iris resource algebras we remark that the above can be realized using the resource algebra $\textsc{Auth}(\mathscr{P}(Loc))$ and the following definitions:

$$\ell_n \dashrightarrow \gamma_m \triangleq \boxed{\circ\,\{\ell_n\}}^{\gamma_m} \qquad \gamma_n \Mapsto \ell_n \triangleq \exists s.\, \boxed{\bullet\, s}^{\gamma_n} * \mathop{\Huge *}_{\ell_m \in s} \ell_m \rightsquigarrow \ell_n$$

Here $\mathscr{P}(A)$ denotes the resource algebra of sets of A, with union as the operation, and the core being the identity function.

5.3 The Invariant

The top-level invariant in Figure 13 is parameterized by a value relation, τ_i, and the values that the implementation and specification consist of. It states the existence of two mathematical lists xs_{i} and xs_{s} that, through I_{MS} and I_{CG}, are related to the physical representation of each queue. The big separating conjunction relates the lists pair-wise by τ_i. This way of relating the implementation and specification is arguably simpler than the approach used in [12, 18], which would have intermingled the physical representations of the two queues with the pair-wise relatedness of the elements in the queues.

I_{CG} is as previously seen and I_{MS} states the existence of ℓ_s, ℓ_t, and ℓ_l and ties the abstract nodes to these. It contains the points-to predicates for the three mutable locations in the queue. It states that the sentinel can reach the abstract tail: $\ell_s \dashrightarrow \gamma_t$. This knowledge is key to proving the **else** branch in dequeue starting on line D13, which we previously discussed. In fact, the reason why the check on D6 ensures that the queue is empty is exactly that the tail pointer can not fall behind the sentinel pointer. Additionally, $\ell_t \dashrightarrow \gamma_l$ ensures that the tail can reach the abstract last node. Finally, $\mathrm{isQueue}_{\mathrm{MS}}$ relates the linked list to the mathematical list xs_{i}.

Note how the only non-persistent things in I_{MS} are the three points-to predicates and the resource tieing the abstract nodes to the physical nodes. Clearly, these can not be persistent. Hence, our invariant precisely captures and separates the changing parts of the MS-queue from the unchanging parts.

Before moving on to the refinement proof, we demonstrate how the invariant and abstract reachability is used by proving a lemma which is to be used whenever the MS-queue attempts to swing the tail pointer forward.

Lemma 5.1. *Swing tail pointer forward.*

$$\frac{\boxed{I(\dots)}^{\iota} \qquad \ell_n \rightsquigarrow \ell_m \qquad \forall v.\, \models K[\, v\,] \precsim e : \alpha}{\models K[\, \textit{\textbf{CAS}}\; \ell_{\rightarrow t}\; \ell_n\; \ell_m\,] \precsim e : \alpha}$$

Top-level invariant

$$I(\tau_i, \ell_{\rightarrow s}, \ell_{\rightarrow t}, \ell_{cg}, lk) \triangleq \exists xs_i, xs_s.\ \mathrm{I}_{\mathrm{MS}}(\ell_{\rightarrow s}, \ell_{\rightarrow t}, xs_i) * \mathrm{I}_{\mathrm{CG}}(\ell_{cg}, lk, xs_s)\ * \mathop{\circledast}_{(x_i,x_s)\in(xs_i,xs_s)} \tau_i(x_i, x_s)$$

Invariant for the MS-queue

$$\begin{aligned}\mathrm{I}_{\mathrm{MS}}(\ell_{\rightarrow s}, \ell_{\rightarrow t}, xs_i) \triangleq \exists \ell_s, \ell_{s\rightarrow}, \ell_t, \ell_{t\rightarrow}, \ell_l, \ell_{l\rightarrow}.\ & \ell_{\rightarrow s} \hookrightarrow_{\mathrm{i}} \ell_s * \ell_{\rightarrow t} \hookrightarrow_{\mathrm{i}} \ell_t * \mathrm{isQueue}_{\mathrm{MS}}(\ell_{l\rightarrow}, \ell_{s\rightarrow}, xs_i) * \\ & \gamma_s \Mapsto \ell_s * \ell_s \hookrightarrow_{\mathrm{i}}^{\square} \mathsf{some}\ (-, \ell_{s\rightarrow}) * \ell_s \dashrightarrow \gamma_t * \\ & \gamma_t \Mapsto \ell_t * \ell_t \hookrightarrow_{\mathrm{i}}^{\square} \mathsf{some}\ (-, \ell_{t\rightarrow}) * \ell_t \dashrightarrow \gamma_l * \\ & \gamma_l \Mapsto \ell_l * \ell_l \hookrightarrow_{\mathrm{i}}^{\square} \mathsf{some}\ (-, \ell_{l\rightarrow}) * \ell_{l\rightarrow} \hookrightarrow_{\mathrm{i}} \ell_n * \ell_n \hookrightarrow_{\mathrm{i}}^{\square} \mathsf{none}\end{aligned}$$

$$\mathrm{isQueue}_{\mathrm{MS}}(\ell_{l\rightarrow}, \ell_{\rightarrow n}, \quad []) \triangleq \ell_{l\rightarrow} = \ell_{\rightarrow n}$$

$$\mathrm{isQueue}_{\mathrm{MS}}(\ell_{l\rightarrow}, \ell_{\rightarrow n},\ x :: xs) \triangleq \exists \ell_n, \ell_{n\rightarrow}.\ \ell_{\rightarrow n} \hookrightarrow_{\mathrm{i}}^{\square} \ell_n * \ell_n \hookrightarrow_{\mathrm{i}}^{\square} \mathsf{some}\ (\mathsf{some}\ x, \ell_{n\rightarrow}) * \mathrm{isQueue}_{\mathrm{MS}}(\ell_{l\rightarrow}, \ell_{n\rightarrow}, xs)$$

Figure 13. The invariant and auxiliary definitions.

Proof. We apply REL-CAS-L and open the invariant. Since the invariant contains $\ell_{\rightarrow t} \hookrightarrow \ell_t$ for some ℓ_t we offer the witness ℓ_t. If the **CAS** fails we can simply close the invariant again. If the **CAS** succeeds we know that $\ell_n = \ell_t$ and we now get $\ell_{\rightarrow t} \hookrightarrow \ell_m$. When we close the invariant we supply ℓ_m as the witness for ℓ_t. To do that we have to show

$$\gamma_t \Mapsto \ell_m * \ell_m \hookrightarrow_{\mathrm{i}}^{\square} \mathsf{some}\ (-, \ell_{m\rightarrow}) * \ell_m \dashrightarrow \gamma_l$$

The middle conjunction follows from $\ell_n \rightsquigarrow \ell_m$. We have $\gamma_t \Mapsto \ell_n$ and $\ell_n \rightsquigarrow \ell_m$ which per the last rule in Figure 12 gets us the rest. □

6 Refinement Proof of the MS-queue

We now prove that the MS-queue contextually refines the coarse-grained queue:

$$\models \mathsf{queue}_{\mathrm{MS}} \precsim \mathsf{queue}_{\mathrm{CG}} : \forall \alpha.(1 \to \mathsf{Option}\ \alpha) \times (\alpha \to 1)$$

Since both $\mathsf{queue}_{\mathrm{MS}}$ and $\mathsf{queue}_{\mathrm{CG}}$ are type abstractions we apply REL-TLAM to show that in a context extended with α interpreted using any value relation R. We symbolically execute the code on the left-hand side to the resources:

$$\begin{gathered}\ell_{nil} \hookrightarrow_{\mathrm{i}} \mathsf{none} * \ell_{s\rightarrow} \hookrightarrow_{\mathrm{i}} \ell_{nil} * \\ \ell_s \hookrightarrow_{\mathrm{i}} \mathsf{some}(\mathsf{none}, \ell_{s\rightarrow}) * \ell_{\rightarrow s} \hookrightarrow_{\mathrm{i}} \ell_s * \ell_{\rightarrow t} \hookrightarrow_{\mathrm{i}} \ell_s\end{gathered}$$

From stepping through the right-hand side we get

$$\ell_{list} \hookrightarrow_{\mathrm{s}} \mathsf{none} * \mathsf{isLocked}(lk, \mathsf{False}).$$

Together with ABS-REACH-ALLOC this is enough to establish the invariant. We thus now have $\boxed{I(\tau_i, \ell_{\rightarrow s}, \ell_{\rightarrow t}, \ell_{cg}, lk)}^{\iota}$ in the context.

Both sides step to a pair and we apply the structural rule REL-PAIR. We are then required to show that the fine-grained dequeue and enqueue are logical refinements of their coarse-grained counterparts. We do this in the next two sections.

6.1 Dequeue

We are to show the logical refinement:

$$\begin{aligned}[\alpha := R] \models\ & \mathsf{dequeue}_{\mathrm{MS}}\ \ell_{\rightarrow s}\ \ell_{\rightarrow t} \\ & \precsim \mathsf{dequeue}_{\mathrm{CG}}\ lk\ \ell_{CG} : 1 \to \mathsf{Option}\ \alpha.\end{aligned}$$

Since both sides are functions we use REL-REC and have to show that for any two values v_1 and v_2, where $[\![1]\!]_{\Delta}(v_1, v_2)$, it is the case that the left-hand side applied to v_1 is related to the right-hand side applied to v_2. Since v_1 and v_2 are related at the type 1 they must both be equal to the unit value (). Hence we are to show

$$\begin{aligned}[\alpha := R] \models\ & \mathsf{dequeue}_{\mathrm{MS}}\ \ell_{\rightarrow s}\ \ell_{\rightarrow t}\ () \\ & \precsim \mathsf{dequeue}_{\mathrm{CG}}\ lk\ \ell_{CG}\ () : \mathsf{Option}\ \alpha.\end{aligned}$$

As the left-hand side is a recursive function we apply the LÖB rule. This gives us the induction hypothesis that the refinement holds for any recursive calls. We then apply structural rules to symbolically execute the left implementation until we arrive at the first load:

$$\mathsf{sent} = \ !\,\ell_{\rightarrow s}$$

The yellow background indicates the expression currently being symbolically executed and which we open the invariant around. We open the invariant and from the points-to predicate for $\ell_{\rightarrow s}$ we know that the load steps to some ℓ_s and that we can assume the following persistent propositions for some $\ell_{s\rightarrow}$ and v:

$$\ell_s \hookrightarrow_{\mathrm{i}}^{\square} \mathsf{some}\ (v, \ell_{s\rightarrow}) * \ell_s \dashrightarrow \gamma_s * \ell_s \dashrightarrow \gamma_t * \ell_s \dashrightarrow \gamma_l \qquad (1)$$

On the next line, the tail is loaded.

$$\mathsf{tail} = \ !\,\ell_{\rightarrow t}$$

By opening the invariant, we can conclude that the load evaluates to some ℓ_t. We know that ℓ_s can reach the current tail ($\ell_s \dashrightarrow \gamma_t$ in Eq. (1)) and that ℓ_t is the current tail ($\gamma_t \Mapsto \ell_t$ from the invariant) hence per ABS-REACH-CONCR we get $\ell_s \rightsquigarrow \ell_t$.

On the next line (D4a) ℓ_s is read:

$$\mathsf{toNext} = \pi_2\ (\mathsf{getValue}\ \boxed{!\,\ell_s}\,)$$

We can evaluate this, without opening the invariant, using the points-to predicate from Eq. (1). Thus, the load evaluates to some $(v, \ell_{s\to})$. With this information, we can symbolically execute the getValue and the projection.

We then arrive at the creation of the prophecy variable at line D4b. Using rel-newproph-l we get the prophecy assertion $\mathsf{Proph}_1(p, v)$. Since the prophecy variable is resolved with $!\,\mathsf{toSent}$ on line D5, the value v is, intuitively, equal to the result of that load. Hence, whether or not v is equal to ℓ_s, determines the outcome of the check on line D5. If they are equal, we will be able to show that the check succeeds, and otherwise, the check will fail. We consider these two cases separately. In the latter case, where $v \neq \ell_s$, dequeue restarts and we only have to show that the execution up to the recursive call on the last line is safe. This is straightforward so we consider only the first case where $v = \ell_s$.

We proceed to the next load:

$$\mathsf{next} = \boxed{!\,\ell_{s\to}}$$

This load reads the pointer *out* of the *sentinel*. Intuitively, if this leads to none then the queue must be empty and the pointer read is the mutable pointer that $\mathsf{enqueue}_{\mathrm{MS}}$ may modify. Hence, if this is the case, this is a linearization point and we must then conclude that the queue is empty.

To do this, we open the invariant and introduce the existentially quantified locations with the names ℓ_s, ℓ_t and ℓ_l as $\ell_{s'}, \ell_{t'}$ and $\ell_{l'}$ respectively. Using $\ell_s \dashrightarrow \gamma_s$ from Eq. (1) and abs-reach-concr we can determine that ℓ_s can reach all these nodes:

$$\ell_s \rightsquigarrow \ell_{s'} * \ell_s \rightsquigarrow \ell_{t'} * \ell_s \rightsquigarrow \ell_{l'} \tag{2}$$

Since ℓ_s can reach $\ell_{l'}$ they are either equal or ℓ_s has a successor node which can reach $\ell_{l'}$.

First case: We have $\ell_s = \ell_{l'}$. The sentinel read earlier is equal to the current tail. Then all the nodes in Eq. (2) reachable from ℓ_s are reachable from $\ell_{l'}$. But, $\ell_{l'}$ has no successors ($\ell_{l'\to}$ points to none) hence any node it can reach must be itself:

$$\ell_s = \ell_t = \ell_{l'} = \ell_{s'} \tag{3}$$

Per Mapsto-agree-□ this implies that $\ell_{s\to} = \ell_{l'\to}$. We thus find that the pointer being loaded is $\ell_{l'\to}$ and the points-to predicates

$$\ell_{l'\to} \hookrightarrow_{\mathrm{i}} \ell_{nil} * \ell_{nil} \hookrightarrow_{\mathrm{i}}^{\square} \mathsf{none}$$

are in the invariant. Hence the load results in ℓ_{nil}.

By combining the above with the following fact

$$\begin{aligned}&\mathrm{isQueue}_{\mathrm{MS}}(\ell_{s\to}, \ell_{s\to}, xs) \twoheadrightarrow\\ &\quad \ell_{s\to} \hookrightarrow_{\mathrm{i}} \ell_{nil} \twoheadrightarrow \ell_{nil} \hookrightarrow_{\mathrm{i}}^{\square} \mathsf{none} \twoheadrightarrow xs = [].\end{aligned}$$

we conclude that $xs_{\mathrm{i}} = []$ and hence also (from the big separating conjunction in I) that $xs_{\mathrm{s}} = []$. Using $xs_{\mathrm{s}} = []$ we can now apply $\text{dequeue}_{\mathrm{CG}}\text{-nil-r}$. After this our goal is to show the refinement:

$$[\alpha := R] \models K[\ !\,\ell_{s\to}\ ] \precsim \mathsf{none} : \mathsf{Option}\ \alpha.$$

We must show that the left-hand side steps to none which we can do as follows: On line D5 we know that the check in the if-statement is **true** since we know that the prophecy variable is resolved to ℓ_s. Hence symbolic execution proceeds to line D6 where ℓ_s is compared to ℓ_t. From Eq. (3) we know that these are equal. On line D7 the location ℓ_{nil} is loaded; it points-to none and thus the function returns none on line D8.

Second case: There exists a node ℓ_n for which we have

$$\ell_{s\to} \hookrightarrow_{\mathrm{i}}^{\square} \ell_n * \ell_n \hookrightarrow_{\mathrm{i}}^{\square} \mathsf{some}\ (v, \ell_{n\to}) * \ell_n \rightsquigarrow \ell_{l'}.$$

The load evaluates to ℓ_n and we close the invariant.

On line D6 the location ℓ_s is compared to ℓ_t and we case on whether or not these locations are equal:

Case $\ell_s = \ell_t$***:*** The **if**-statement succeeds, we step to D7 which loads ℓ_n and thus evaluates to a some. Therefore the **match** takes the second branch to D10:

$$\boxed{\mathbf{CAS}\ \ell_{\to t}\ \ell_t\ \ell_n}\,;\mathsf{loop}\ ()$$

Here we apply Lemma 5.1, and for the last expression we apply the induction hypothesis.

Case $\ell_s \neq \ell_t$***:*** We step to D13 where dequeue attempts to swing the sentinel pointer forward:

$$\mathbf{if}\ \boxed{\mathbf{CAS}\ \ell_{\to s}\ \ell_s\ \ell_n}$$

We know that the **CAS** is safe since the invariant contains the points-to predicate $\ell_{\to s} \hookrightarrow_{\mathrm{i}} \ell_{s'}$ for some $\ell_{s'}$.

If the **CAS** fails we have not changed anything and can simply close the invariant, step to D15, and apply the induction hypothesis.

If the **CAS** succeeds then $\ell_s = \ell_{s'}$ and this is a linearization point. After the **CAS** we have $\ell_{\to s} \hookrightarrow_{\mathrm{i}} \ell_n$. Since ℓ_s is equal to $\ell_{s'}$ the pointer out of $\ell_{s'}$ must be equal to $\ell_{s\to}$. As such we have $\mathrm{isQueue}_{\mathrm{MS}}(\ell_{s\to}, \ell_{l\to}, xs_{\mathrm{i}})$ from the invariant for some xs_{i}.

If xs_{i} was $[]$ then ℓ_s would be equal to the last node, which points to none. But, this is in contradiction with the knowledge that ℓ_s is succeeded by ℓ_n. Hence xs_{i} cannot be $[]$. Thus there exists x_i and xs'_i such that $xs_{\mathrm{i}} = x_i :: xs'_i$; and x_s and xs'_s such that $xs_{\mathrm{s}} = x_s :: xs'_s$. For these:

$$\tau_i(x_i, x_s) * \mathop{\Large *}_{(x_i, x_s) \in (xs'_i, xs'_s)} \tau_i(x_i, x_s)$$

Moreover, x_i must be exactly the value in the node ℓ_n (i.e., $v = \mathsf{some}\ x_i$).

With the knowledge that the list is non-empty we can use $\text{dequeue}_{\mathrm{CG}}\text{-cons-r}$ after which we get $\mathrm{I}_{\mathrm{CG}}(\ell_{cg}, lk, xs'_{\mathrm{i}})$ and must show the refinement:

$$[\alpha := R] \models_{\mathcal{E}} K[\,\mathbf{true}\,] \precsim \mathsf{some}\ x_s : \tau$$

When we close the invariant we offer ℓ_n as a witness for the existentially quantified variable ℓ_s. To do this we must show $\gamma_s \Mapsto \ell_n$ and $\ell_n \dashrightarrow \gamma_t$—this is fairly easy.

After the **CAS** we arrive at D14. We know that the load evaluates to some (some $x_s, \ell_{n\to}$). Hence the entire expression on line D14 steps to some x_s and we are to show

$$[\alpha := R] \models_{\mathcal{E}} \mathsf{some}\ x_i \precsim \mathsf{some}\ x_s : \tau$$

which we can do because we have $\tau_i(x_i, x_s)$.

6.2 Enqueue

To conclude the proof we show refinement of enqueue:

$$[\alpha := R] \models \mathsf{enqueue}_{\mathrm{MS}}\ \ell_{\to t} \precsim \mathsf{enqueue}_{\mathrm{CG}}\ lk\ \ell_{list} : \alpha \to 1.$$

As both sides of the refinement are lambda-values we must show that these are related when applied to any two values, x_i and x_s, related by τ_i.

We first step over the construction of the new node on line E1. This gives us the resources:

$$\ell_n \hookrightarrow_i \mathsf{some}\ (\mathsf{some}\ x_i, \ell_{n\to}) * \ell_{n\to} \hookrightarrow_i \ell_{nil} * \ell_{nil} \hookrightarrow_i \mathsf{none}$$

Line E4 is an application of a recursive function. We therefore apply the LÖB rule as we did in the proof of dequeue.

To step over the load of $\ell_{\to t}$ on line E5 we open the invariant which contains the points-to predicate $\ell_{\to t} \hookrightarrow_i \ell_t$ for some ℓ_t. The load evaluates to ℓ_t and when we close the invariant we keep the following persistent knowledge:

$$\ell_t \dashrightarrow \gamma_l * \ell_t \hookrightarrow_i^{\square} \mathsf{some}\ (v, \ell_{t\to}), \tag{4}$$

for some v and $\ell_{t\to}$. The persistent points-to predicate for ℓ_t is used for the load on the next line, E6a. Since its contents match the operations applied to it, we can symbolically execute the rest of the line, and toNext is assigned to the value $\ell_{t\to}$.

The next line (E6b) loads $\ell_{t\to}$ and we open the invariant again. The invariant contains $\gamma_l \Mapsto \ell_l$ for some ℓ_l. By using ABS-REACH-CONCR we get $\ell_t \rightsquigarrow \ell_l$. We case on whether or not ℓ_t is equal to ℓ_l.

First case, $\ell_t = \ell_l$***:*** . We rewrite with the equality in the points-to predicate in Eq. (4) and get $\ell_l \hookrightarrow_i^{\square}$ some $(v, \ell_{t\to},)$. From the invariant we have $\ell_l \hookrightarrow_i^{\square}$ some $(v', \ell_{l\to})$ and thus, by MAPSTO-AGREE-$\square$, we get $\ell_{t\to} = \ell_{l\to}$. From the invariant we further have

$$\ell_{l\to} \hookrightarrow_i \ell_{nil} * \ell_{nil} \hookrightarrow_i^{\square} \mathsf{none} \tag{5}$$

Hence we can conclude that the load evaluates to ℓ_{nil}. We close the invariant.

Symbolic execution continues to line E7. On this line $\ell_{\to t}$ is loaded again. We have already seen how the invariant ensures that such a load is safe. The newly read value is then compared to the old value read at line E5. If these are not equal symbolic execution proceeds to line E14 where we can conclude the proof by applying the induction hypothesis. If they are equal execution proceeds to line E8 where ℓ_{nil} is loaded. We use the points-to predicate from Eq. (5) and conclude that the load evaluates to none.

Therefore the **match** takes the first branch to the **CAS** on line E9:

$$\mathbf{if}\ \boxed{\mathbf{CAS}\ \ell_{t\to}\ \ell_{nil}\ \ell_n}$$

To show that the **CAS** is safe we must have a points-to predicate for $\ell_{t\to}$. We can open the invariant and get a points-to predicate $\ell_{l'\to} \hookrightarrow_i \ell_{nil}$ for some $\ell_{l'\to}$. Intuitively, if the **CAS** succeeds it is because $\ell_{t\to}$ is still the last node in the linked list and in that case $\ell_{t\to}$ is equal to $\ell_{l'\to}$.

This is where we apply our novel REL-CAS-L, which is quite subtle. This rule asks us to supply a witness which we must later show that ℓ_t points-to. To find such a witness observe that ℓ_t can reach $\ell_{l'}$. If they are equal then $\ell_{t\to}$ is equal to $\ell_{l'\to}$ and $\ell_{t\to}$ points to ℓ_{nil}. If they are not equal then $\ell_{t\to}$ must point to some other node. In both cases $\ell_{t\to}$ points to something, but in the first case the reasoning relies on the resource $\ell_{l'\to} \hookrightarrow_i \ell_{nil}$. Hence by giving up this resource we can conclude that there exists some ℓ_m such that

$$\begin{aligned} &\exists \ell_{m\to}.\ \ell_{t\to} \hookrightarrow_i^{\square} \ell_m * \ell_m \hookrightarrow_i^{\square} \mathsf{some}\ (-, \ell_{m\to}) * \ell_{l\to} \hookrightarrow_i \ell_{nil} \\ &\quad \vee \ell_{t\to} \hookrightarrow_i \ell_m * \ell_t = \ell_{l'} * \ell_m = \ell_{nil}. \end{aligned} \tag{6}$$

We offer this ℓ_m as a witness. We now have two cases corresponding to whether the **CAS** fails or succeeds and to the disjunction in REL-CAS-L.

CAS succeeds. If the **CAS** succeeds then this is a linearization point. We must show the *full* points-to predicate (not just a persistent points-to) for $\ell_{t\to}$, but we only have the full points-to predicate in one of the disjuncts in Eq. (6). But, from the rule we can assume that ℓ_m is equal to ℓ_{nil}, which points to none. This leads to a contradiction in the first disjunct in Eq. (6) which states that ℓ_m points to a some. We can therefore assume the last disjunct. This does not only give us the full points-to predicate we need, it also tells us that ℓ_t is equal to the current last node $\ell_{l'}$ which is important to ensure that our change affects the queue correctly. Notice the subtlety involving equality, used to conclude that we had the full points-to predicate. Since we have now changed $\ell_{l'\to}$ we can use $\mathrm{isQueue}_{\mathrm{MS}}(\ell_{l'\to}, \ell_{s\to}, xs_i)$ to show $\mathrm{isQueue}_{\mathrm{MS}}(\ell_{l'\to}, \ell_{s\to}, xs_i +\!\!+ [x])$. We have changed the last node from ℓ_t into ℓ_n. So we need to change $\gamma_l \Mapsto \ell_t$ into $\gamma_l \Mapsto \ell_n$. Clearly $\ell_t \rightsquigarrow \ell_n$, so we can use ABS-REACH-ADVANCE to achieve this.

Since this is the linearization point we use ENQUEUE$_{\mathrm{CG}}$-R to step the specification forward. We then have everything needed to close the invariant.

We continue to E17 where we apply Lemma 5.1 to show that the attempt at advancing the tail pointer is safe. The final expression is then () which matches the right-hand side at this point.

CAS fails. In this case we, can assume that $\ell_{l'} \neq \ell_l$. Following the rule REL-CAS-L we have to provide either a persistent or fractional points-to predicate for $\ell_{t\rightarrow}$. And from Eq. (6) we know that we have one of these. We therefore consider each case in the disjunction and pick the corresponding case to show. This shows that the **CAS** is safe, and since nothing changed, it is trivial to close the invariant again. Execution steps to E11 where we apply the induction hypothesis.

Second case, $\ell_t \neq \ell_l$: . In this case, the tail pointer was lagging behind when we read it and there exists a node ℓ_m for which we have

$$\ell_{t\rightarrow} \hookrightarrow^{\square}_{\mathsf{i}} \ell_m * \ell_m \hookrightarrow^{\square}_{\mathsf{i}} \mathsf{some}\ (v', \ell_{m\rightarrow}) * \ell_m \rightsquigarrow \ell_l.$$

Hence the load evaluates to ℓ_m. We close the invariant.

Line E7 is handled as before. The load is safe, and if the two locations are not equal we apply the induction hypothesis at line E43. If the locations are equal we proceed to line E8 where ℓ_m is loaded. Since ℓ_m points to a some we step to E13. At E13 we apply Lemma 5.1 and then the induction hypothesis.

7 Consistent Snapshots Can Be Omitted

Recall the consistent snapshots in dequeue (line D5) and enqueue (line E7). The consistent snapshots are meant to solve the ABA problem by ensuring that the values read are still up-to-date. However, with the insights gained from our formal proof, it becomes evident that these snapshots are actually *not* needed for correctness: from the way we have constructed the invariant we do not need to use the information gained from these checks. This is because the instance of the ABA problem that the consistent snapshot solves does in fact not occur in a garbage collected setting. And since the semantics of the language of our implementation, HeapLang, models a garbage collected language, we can formally prove that the atomic snapshots are not needed.

In the Coq formalization of our proofs, we have shown that the MS-queue without the consistent snapshots still contextually refines the coarse-grained queue. We have also shown that the coarse-grained queue refines the MS-queue both with and without the consistent snapshots. This implies that the coarse-grained queue is contextually *equivalent* to both queues, and, per transitivity of contextual refinement, that the MS-queue with consistent snapshots is contextually equivalent to one without.

We speculate that omitting the consistent snapshots may result in better performance as dequeue may still succeed even if the consistent snapshot fails. Hence this can lead to earlier success. As one can see in our Coq formalization, for the refinement proof of the MS-queue without the consistent snapshots it is not necessary to use prophecy variables in the proof.

8 Lagging-Tail MS-queue

Our Coq formalization also contains a HeapLang implementation and a refinement proof for what we name the *lagging-tail* MS-queue. It resembles how the queue included in the Java standard library works and is a slightly more realistic version of the queue covered in [18]. This variant is quite different from the original MS-queue in that it allows the tail pointer to lag behind arbitrarily, a change affecting both how dequeue and enqueue works: Dequeue can no longer rely on the sentinel being able to reach the tail and enqueue must read the tail pointer and, to account for the lagging tail, then iterate through the linked list until it finds the last node. While this is in many ways a simpler algorithm to prove correct, we find it remarkable that our notion of reachability also suffices to prove contextual refinement for this, very different, variant with only a very small change to the invariant. As the tail pointer may lag behind arbitrarily, it may, in particular, be further behind than even the sentinel pointer. Hence to prove contextual refinement for this variant we can no longer include $\ell_s \dashrightarrow \gamma_t$ in the invariant. However, by simply changing this part to $\ell_s \dashrightarrow \gamma_l$, we can prove refinement of the variant. No other changes are required to the invariant!

9 Defining the Persistent Points-To Predicate

This section describes how we implement the persistent points-to predicate. In Iris, Hoare triples, the weakest precondition, and the points-to predicate are not primitives in the logic. Instead, they are defined *inside the logic*, using what is called the Iris *base logic*. Hence we can implement the persistent points-to predicate entirely inside Iris, by changing the definitions that constitute the weakest precondition. An advantage of this approach is that soundness of the rules for the persistent points-to predicate follows directly from soundness of the Iris base logic.

The biggest challenge in adding the persistent points-to predicate is to ensure that it satisfies Mapsto-intro-□. The existing points-to predicate is defined as ownership of some ghost state. Hence to make this rule true we need to use a resource algebra (RA) that supports a frame-preserving update from the ghost state owned by the normal points-to predicate to the ghost state owned by the persistent points-to predicate. We solve this by introducing the *discardable fractions RA*.

For space reasons, in the rest of this section we assume that the reader is familiar with ghost state and resource algebras in Iris. For the details, we refer to [9].

Encoding of the heap. To extend Iris as described we need to change two existing definitions: *heapCtx* and $\hookrightarrow_q$. The former is a predicate on heaps

$$heapCtx : (Loc \xrightarrow{\text{fin}} Val) \to \mathsf{iProp}.$$

which is part of the state interpretation used in the definition of the weakest precondition. For every step of execution, starting in a heap σ and ending in heap σ', $heapCtx(\sigma)$ holds before and $\Rrightarrow heapCtx(\sigma')$ holds after the step.

In the current version of Iris, *heapCtx* is defined using the RA

$$\textsc{Auth}(Loc \xrightarrow{\text{fin}} (\mathbb{Q}_{01} \times \textsc{Ag}(Val))), \tag{7}$$

and the following definitions[2]:

$$heapCtx(\sigma) = \boxed{\bullet\,\sigma}^{\gamma_{\text{heap}}} \qquad \ell \hookrightarrow_q v = \boxed{\circ\,[\ell \leftarrow (q, \text{ag}(v))]}^{\gamma_{\text{heap}}}$$

We note that $\ell \hookrightarrow_q v$ is not persistent since $\mathbb{Q}_{01}$ has no core. Updates to the heap are possible since $1 \in \mathbb{Q}_{01}$ is exclusive (it has no frame).

Recall that we want Mapsto-intro-□ to hold *without depending on heapCtx*. This is because *heapCtx* is internal to the definition of weakest precondition and not exposed to clients of it. We therefore need to use an RA that makes it possible to make a frame-preserving update from the ghost state owned by $\hookrightarrow_q$ to the ghost state owned by $\hookrightarrow_\square$. The core should be undefined for the former while defined for the latter. We define such an RA in the next section. But, even with such an RA we have the problem that $\hookrightarrow$ denotes ownership of a fragment, and with the authoritative RA it is not clear how to make a suitable frame-preserving update from a fragment. We therefore also need to introduce a generalized authoritative RA.

Discardable fractions RA. We introduce the RA of *discardable fractions*, which is a generalization of the normal fractional RA. Whereas elements of the fractional RA denote *ownership* over some strictly positive fraction, elements of the discardable fractional RA can additionally denote *knowledge* about a fraction having been discarded.

Let $\mathbb{Q}_{>0}$ denote the set of strictly positive rationals. The carrier for the RA is:

$$\textsc{DFrac} \triangleq \mathsf{own}(q) \mid \mathsf{disc}(p) \mid \mathsf{both}(q, p) \qquad q, p \in \mathbb{Q}_{>0}$$

One should think of this as pairs where one, *but not both*, of the values might be absent. The element $\mathsf{own}(q)$ is equivalent to an element of the normal fractional RA and the element $\mathsf{disc}(p)$ denotes the knowledge that the fraction p has been discarded.

The valid elements are those where the sum of the two numbers are less than or equal to 1:

$$\mathcal{V}(\mathsf{own}(p)) \triangleq p \leq 1 \qquad \mathcal{V}(\mathsf{disc}(q)) \triangleq q \leq 1$$
$$\mathcal{V}(\mathsf{both}(q, p)) \triangleq q + p \leq 1$$

The operation adds together the owned fractions and takes the maximum of the fractions known to be discarded. We do not specify all cases in the operation, the remaining cases are determined by the requirement that the operation is commutative and associative.

$$\begin{aligned} \mathsf{disc}(p) \cdot \mathsf{disc}(p') &\triangleq \mathsf{disc}(\max(p, p')) \\ \mathsf{own}(q) \cdot \mathsf{own}(q') &\triangleq \mathsf{own}(q + q') \\ \mathsf{own}(q) \cdot \mathsf{disc}(p) &\triangleq \mathsf{both}(q, p) \end{aligned}$$

The core of an element is the discarded part of the element if any. This ensures that knowledge about discarded fractions is persistent.

$$|\mathsf{disc}(p)| = \mathsf{disc}(p) \quad |\mathsf{own}(q)| = \bot \quad |\mathsf{both}(q, p)| = \mathsf{disc}(p)$$

We now have the following frame-preserving update.

Lemma 9.1. *Discarding is possible: $own(q) \rightsquigarrow disc(q)$.*

Proof. Suppose $\mathsf{own}(q) \cdot \mathsf{both}(q', p')$ is valid. Then $q + q' + p' \leq 1$, which implies that $q' + \max(q + p') \leq 1$ showing that $\mathsf{disc}(q) \cdot \mathsf{both}(q', p')$ is valid. The remaining cases are similar. □

Heap RA. We would now like to replace the use of the fractional RA in Eq. (7), the RA currently used for the heap, with the discardable fractional RA. However, this alone is not enough because, as mentioned, the authoritative RA does make it possible to make the frame-preserving update from a fragment that we need.

We therefore need a slightly generalized variant of the authoritative RA that allows us to update the discardable fraction in fragments. For RA's A and B and a function $\pi : B \to A$ we define

$$\begin{aligned} \textsc{PAuth}(A, B, \pi) &= \textsc{Ex}(A)^? \times B \\ \mathcal{V}((\bot, b)) &= \mathcal{V}(b) \\ \mathcal{V}((a, b)) &= \mathcal{V}(a) \wedge \mathcal{V}(b) \wedge \pi(b) \preccurlyeq a \\ (a, b) \cdot (a', b') &= (a \cdot a', b \cdot b') \\ |(a, b)| &= \begin{cases} (\bot, |b|) & \text{if } |b| \neq \bot \\ \bot & \text{otherwise} \end{cases} \end{aligned}$$

The full and fragmental view is defined as usual.

$$\bullet\, a \triangleq (a, \varepsilon) \qquad\qquad \circ\, b \triangleq (\bot, b)$$

For this construction to satisfy the laws of a RA π must be expansive with respect to the inclusion order.

The difference between this construction and the normal authoritative RA is that the authoritative and fragmental view can contain two different RA's and that in the definition of validity $\pi(b)$, and not b itself, should be included in a.

To model the heap we then instantiate the above construction by using

$$\textsc{PAuth}(Loc \xrightarrow{\text{fin}} \textsc{Ag}(Val), Loc \xrightarrow{\text{fin}} (\textsc{DFrac} \times \textsc{Ag}(Val)), \pi_2).$$

[2]This is simplified—but covers what is relevant for our purpose.

The definitions for the heap are then

$$heapCtx(\sigma) \triangleq \boxed{\bullet\, \sigma}^{\gamma_{\text{heap}}}$$
$$\ell \hookrightarrow_q v \triangleq \boxed{\circ\, [\ell \leftarrow (\mathsf{own}(q), v))]}^{\gamma_{\text{heap}}}$$
$$\ell \hookrightarrow_\square v \triangleq \exists p.\, \boxed{\circ\, [\ell \leftarrow (\mathsf{disc}(p), v)]}^{\gamma_{\text{heap}}}$$

This ensures that the fraction in the fragment is independent of the full authoritative view and hence that it can be updated without the full authoritative view.

Lemma 9.2. *If $q, q' \in$ DFrac and $q \rightsquigarrow q'$ then* $\circ\, [k \leftarrow (q, v)] \rightsquigarrow \circ\, [k \leftarrow (q', v)]$.

Finally, from Lemma 9.1 and Lemma 9.2 we have the frame-preserving update

$$\circ\, [\ell \leftarrow (\mathsf{own}(q), v))] \rightsquigarrow \circ\, [\ell \leftarrow (\mathsf{disc}(p), v)]$$

and can thus show Mapsto-intro-□.

10 Related Work

We now discuss related work that has not already been treated in the paper. The only related work that directly shows contextual refinement is the already mentioned pen-and-paper proof by Turon et. al. However, they only consider a simplification of the less challenging lagging-tail MS-queue. Their approach relies on assigning to each node a *state* in a state transition system. However, they have no notion of reachability, which appears to be necessary for reasoning about the original MS-queue. And since reachability is a *relationship* between two nodes and not a state of one particular node, it is not clear how to extend their approach to the MS-queue. Our approach on the other hand applies to both the MS-queue and the lagging-tail MS-queue.

We now cover related work that shows *linearizability* of the MS-queue. Doherty *et al.* proved that a slightly modified MS-queue is linearizable by using a simulation proof formalized in the PVS proof system [5]. Their simulation proof makes use of both a forward simulation and a backwards simulation; this is comparable to our use of prophecy variables. They make several changes to the queue which they argue improve performance. Their changes preserve the future dependent linearization point, but they also remove the check on line D6, which we found challenging in our proof. Schellhorn *et al.* later showed that backwards simulation suffices to show linearizeability of the MS-queue [17].

Vafeiadis proposed an automatic verification procedure for proving linearizability for first-order programs [19]. His approach handles certain non-fixed linearization points, namely those that are *pure*, meaning that the linearization points do not change the state of the queue. The non-fixed linearization point in dequeue in the MS-queue is pure, as dequeueing an element from an empty queue does not change the state of the queue. Vafeiadis's approach depends on this to obtain a verification procedure for proving linearizability which can handle the MS-queue. His approach is also based on prophecy variables. As mentioned in the Introduction, this notion of linearizability does not imply contextual refinement for our rich higher-order language. We further remark that ReLoC also supports future dependent linearization points even when these are not pure.

Liang and Feng propose a program logic to verify linearizability [13]. They use their approach to verify an impressive number of concurrent data structures, with the MS-queue being one of them. To handle the non-fixed linearization points they use speculation. This approach is related to prophecy variables and does not rely on annotations in the implementation. The program logic and their verification of the MS-queue are not mechanized.

There exists several other approaches to verifying linearizability which can handle non-fixed linearization points, and which should therefore also be able to verify the MS-queue. For these, we refer to the excellent survey [6].

Related to the persistent points-to predicate, Charguéraud and Pottier showed how to extend separation logic with a general read-only modality [4]. This modality makes it possible to temporarily give read-only access to a points-to predicate, without having to keep track of fractions as one needs to do with the fractional points-to predicate. However, even though they remark that it should be possible to construct a predicate for immutable data, they explicitly do not do that. Their approach is for *temporarily* making locations read-only while ours is for *permanently* making locations read-only.

Acknowledgments

We thank Dan Frumin for useful comments on a draft version of the paper and for answering several of the first author's questions regarding the Coq implementation of ReLoC. We also thank the anonymous reviewers for their helpful comments and suggestions. This research was supported in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation.

References

[1] Ales Bizjak and Lars Birkedal. 2018. On Models of Higher-Order Separation Logic. In *Proceedings of the Thirty-Fourth Conference on the Mathematical Foundations of Programming Semantics, MFPS 2018, Dalhousie University, Halifax, Canada, June 6-9, 2018 (Electronic Notes in Theoretical Computer Science)*, Sam Staton (Ed.), Vol. 341. Elsevier, 57–78. https://doi.org/10.1016/j.entcs.2018.03.016

[2] Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. 2005. Permission accounting in separation logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, Jens Palsberg and Martín Abadi (Eds.). ACM, 259–270. https://doi.org/10.1145/1040305.1040327

[3] John Boyland. 2003. Checking Interference with Fractional Permissions. In *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings.* 55–72. https://doi.org/10.1007/3-540-44898-5_4

[4] Arthur Charguéraud and François Pottier. 2017. Temporary Read-Only Permissions for Separation Logic. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings (Lecture Notes in Computer Science)*, Hongseok Yang (Ed.), Vol. 10201. Springer, 260–286. https://doi.org/10.1007/978-3-662-54434-1_10

[5] Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. 2004. Formal Verification of a Practical Lock-Free Queue Algorithm. In *Formal Techniques for Networked and Distributed Systems - FORTE 2004, 24th IFIP WG 6.1 International Conference, Madrid Spain, September 27-30, 2004, Proceedings (Lecture Notes in Computer Science)*, David de Frutos-Escrig and Manuel Núñez (Eds.), Vol. 3235. Springer, 97–114. https://doi.org/10.1007/978-3-540-30232-2_7

[6] Brijesh Dongol and John Derrick. 2014. Verifying linearizability: A comparative survey. *CoRR* abs/1410.6268 (2014). arXiv:1410.6268 http://arxiv.org/abs/1410.6268

[7] Ivana Filipovic, Peter W. O'Hearn, Noam Rinetzky, and Hongseok Yang. 2010. Abstraction for concurrent objects. *Theor. Comput. Sci.* 411, 51-52 (2010), 4379–4398. https://doi.org/10.1016/j.tcs.2010.09.021

[8] Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2020. ReLoC Reloaded: A Mechanized Relational Logic for Fine-Grained Concurrency and Logical Atomicity. *CoRR* abs/2006.13635 (2020). arXiv:2006.13635 https://arxiv.org/abs/2006.13635

[9] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

[10] Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2020. The future is ours: prophecy variables in separation logic. *Proc. ACM Program. Lang.* 4, POPL (2020), 45:1–45:32. https://doi.org/10.1145/3371113

[11] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: a general, extensible modal framework for interactive proofs in separation logic. *Proc. ACM Program. Lang.* 2, ICFP (2018), 77:1–77:30. https://doi.org/10.1145/3236772

[12] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 205–217. https://doi.org/10.1145/3009837

[13] Hongjin Liang and Xinyu Feng. 2013. Modular verification of linearizability with non-fixed linearization points. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 459–470. https://doi.org/10.1145/2491956.2462189

[14] Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, USA, May 23-26, 1996*, James E. Burns and Yoram Moses (Eds.). ACM, 267–275. https://doi.org/10.1145/248052.248106

[15] Andrzej S. Murawski and Nikos Tzevelekos. 2019. Higher-order linearisability. *J. Log. Algebraic Methods Program.* 104 (2019), 86–116. https://doi.org/10.1016/j.jlamp.2019.01.002

[16] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. 55–74. https://doi.org/10.1109/LICS.2002.1029817

[17] Gerhard Schellhorn, John Derrick, and Heike Wehrheim. 2014. A Sound and Complete Proof Technique for Linearizability of Concurrent Data Structures. *ACM Trans. Comput. Log.* 15, 4 (2014), 31:1–31:37. https://doi.org/10.1145/2629496

[18] Aaron Joseph Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. 2013. Logical relations for fine-grained concurrency. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 343–356. https://doi.org/10.1145/2429069.2429111

[19] Viktor Vafeiadis. 2010. Automatically Proving Linearizability. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings (Lecture Notes in Computer Science)*, Tayssir Touili, Byron Cook, and Paul B. Jackson (Eds.), Vol. 6174. Springer, 450–464. https://doi.org/10.1007/978-3-642-14295-6_40

[20] Simon Friis Vindum and Lars Birkedal. 2020. *Contextual Refinement of the Michael-Scott Queue - Coq Artifact.* https://doi.org/10.5281/zenodo.4317021