Mechanized Verification of a Fine-Grained Concurrent Queue from Facebook's Folly Library

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

Dan Frumin
d.frumin@rug.nl
University of Groningen
Netherlands

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

Abstract

We present the first formal specification and verification of the fine-grained concurrent Multi-Producer-Multi-Consumer Queue algorithm from Facebook's Folly Library (MPMC queue). The MPMC queue is a highly optimized fine-grained concurrent algorithm, which scales to thousands of consumer threads and which has been implemented in C++ at Facebook. We present a slightly more high-level implementation of the MPMC queue algorithm and formally prove that it is a contextual refinement of a simple coarse-grained queue (a property which implies that the MPMC queue is linearizable). We formalize our proof using a combination of the ReLoC relational logic and the Iris program logic and the proof is mechanized in the Coq proof assistant. A key challenge of the MPMC queue is that it has a so-called *external linarization point*, which ReLoC has no support for reasoning about. Thus we extend ReLoC, on on paper and in Coq, with novel support for reasoning about external linearization points.

1 Introduction

It is well-known that it is challenging to program, specify, and verify fine-grained concurrent algorithms, and in recent years we have seen much progress on programs logics for specifying and verifying fine-grained concurrent algorithms, e.g., [5, 8, 12, 16, 20, 24–30].

In this paper, we present the first formal specification and verification of the concurrent Multi-Producer-Multi-Consumer Queue algorithm found in Facebook's open source library Folly (or, simply, the MPMC queue in the rest of the paper).

The MPMC queue is a highly optimized fine-grained concurrent algorithm, which scales to thousands of consumer threads and which has been implemented in C++ at Facebook; the code is available online[1]. The MPMC queue was originally developed by Nathan Bronson for the purpose of connecting two thread pools inside TAO [3], Facebook's distributed data store for their social graph. One of the key ideas used in the algorithm is to decrease the contention found in other lock-free data structures, such as the Michael-Scott queue [21], by striping the queue across q queues of size 1, which, in the words of Bronson, "can themselves be simpler and faster" [2].

We present a high-level implementation of the MPMC queue algorithm in an ML-like language with concurrency primitives, and we formally specify and prove the algorithm correct. A popular correctness criterion for concurrent algorithms is *linearizability* [11], briefly, the property that all operations appear to take place atomically at some point during their execution. Instead of showing linearizability directly, we follow the approach suggested by Turon et al. [27] and show that the MPMC queue *contextual refines* a coarse-grained queue, a stronger property than linearizability [7]. The coarse-grained queue is a straightforward implementation of a queue, using a reference to a functional list which is made thread-safe by guarding each the enqueue and dequeue operations by a shared lock. Informally, the contextual refinement property here means that in any program we may replace uses of a simple coarse-grained concurrent queue with the more efficient MPMC queue, without changing the observable behaviour of the program. More precisely, an expression e contextually refines another expression e', denoted $\Delta; \Gamma \vdash e \precsim_{ctx} e' : \tau$, if for all contexts C of ground type, if $C[e]$ terminates with a value, then there exists an execution of $C[e']$ that terminates with the same value.

To show the contextual refinement property, we use the recently proposed relational logic ReLoC [8, 9] and the Iris program logic [13, 14, 16, 18], on top of which ReLoC is defined. One of the reasons why we have chosen to use ReLoC and Iris for our verification effort is that both of these logics are fully formalized in the Coq proof assistant and also come equipped with a so-called proof mode [17, 19] that allows one to reason formally and interactively in ReLoC and Iris using tactics, much as one does in Coq. The Coq development can be found in the accompanying files.

[1] https://github.com/facebook/folly/blob/master/folly/MPMCQueue.h

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference'17, July 2017, Washington, DC, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

To verify a fine-grained concurrent algorithm, one of the key steps is to identify the *linearization points* of its operations. Intuitively, a linearization point for an operation is the point in the code where the operation "appears to take place". One may divide up linearization points into three classes [6]: fixed, future-dependent, and external. The first version of ReLoC [8] had support for reasoning about fixed linearization points, and ReLoC Reloaded [9] also included support for future-dependent linearization points, through its use of Iris-style prophecy variables [15]. However, ReLoC Reloaded did not include support for reasoning about external linearization points. An operation has an external linearization point, if the linearization point happens during the execution of another operation. As we will explain, it turns out that in some cases the linearization point of the dequeue operation of the MPMC queue is external: it happens within the execution of the enqueue operation. To simplify reasoning about external linearization points, we extend ReLoC with new proof rules (and tactics).

The MPMC queue is implemented using several submodules, in particular so-called turn-sequencers and single-element queues (cf. the idea, mentioned above, of striping the queue across a collection of queues of size 1). We believe that it is a strength of our approach that our contextual refinement proof is similarly modular: our refinement proof makes use of (unary) Hoare-style specifications of the turn-sequencer and the single-element queue. As we will explain, we here leverage that ReLoC, following [26], includes a proof rule that allows one to use Hoare-style specifications, written in the Iris program logic, to simplify reasoning about the left-hand-side in a relational proof [8, Section 7.4].

On a more technical note, we remark that in our specifications of the turn-sequencer and the single-element queue, as well as in the MPMC invariant and the refinement proof, we make use of a, to the best of our knowledge, novel proof pattern involving a resource algebra over *infinite* sets, to keep track of which turns are "still available"; see Section 4. We believe this proof pattern could also be used to simplify reasoning about other algorithms, e.g., a ticket lock.

In summary, we make the following contributions:

- Since the MPMC queue has not been described in the literature before, we give a detailed presentation of it (Section 2).
- We informally analyze the MPMC algorithm and identify the linearization points, and observe that one of them is an external linearization point (Section 3).
- We give Hoare-style specifications for the submodules used in the implementation of the MPMC queue (Section 4).
- We show that the MPMC queue contextually refines a coarse-grained queue (Sections 6 and 8). Our proof is *modular* and makes use of the aforementioned Hoare-style specifications for the submodules.
- We extend ReLoC, both on paper and in Coq, with support for reasoning about external linearization points (Section 7).
- We introduce a novel proof pattern involving an encoding of infinite sets as a resource algebra used for specifying and reasoning about infinite sets of turns and tickets.
- All the results in this paper and two additional examples of algorithms with external linearization points are formalized in the Coq proof assistant (Section 9).

We discuss related and future work in Section 9.

2 The Folly MPMC queue

The MPMC queue is implemented using a *single-element queue*, which in turn is implemented using a *turn sequencer*. We describe the three data-structures, starting with the turn sequencer and proceeding bottom up.

2.1 Turn Sequencer

A turn sequencer is a data structure that implements mutual exclusion by *sequentializing* access to a critical section among threads *ordered* by a monotonically increasing turn. The turn sequencer implementation is shown in Figure 1a.

The turn sequencer provides two operations: wait and complete. These are similar to the acquire and release operations on a lock, but they take an additional natural number as an argument. The natural number specifies which *turn* to wait for or to complete. The turn sequencer guarantees that if a threads waits for the nth turn, then it will only proceed once all the preceding turns have been completed. In order for this to hold, the turn sequencer assumes that its clients never wait for the same turn several times. As such it is the responsibility of clients to manage the turns, *i.e.,* which natural numbers they wait for. Compared to a lock, this places a greater demand on the client, but in return the client is given precise control over the order in which threads run their critical sections.

We implement the turn sequencer as a pointer ts to a number, which represents the current turn. The function wait ts n simply spins on that pointer until its value is equal to n. The implementation of complete ends the current turn by incrementing ts.

2.2 Single-Element Queue

A single-element queue is a queue with capacity 1. Our implementation is shown in Figure 1a. It is a *blocking* queue: if it is empty (full) then any subsequent dequeue (enqueue) is blocked until the queue becomes non-empty (non-full). This implies that both dequeue and enqueue always succeeds.

Similarly to the turn sequencer, the single-element queue's operations take a turn as an argument. The turn argument specifies the order of the operations: an enqueue or dequeue operation is carried out only after all operations with a lower

```
newTS () = ref(0)
complete ts turn = ts ← turn + 1
wait ts turn =
    let turn′ = ! ts in
    if (turn′ = turn) then ()
    else wait ts turn

newSEQ () = (newTS (), ref(None))
enqueueSEQ (ts, r) enqTurn v =
    let turn = enqTurn ∗ 2 in
    wait ts turn;
    r ← Some(v);
    complete ts turn
dequeueSEQ (ts, r) deqTurn =
    let turn = deqTurn ∗ 2 + 1 in
    wait ts turn;
    let v = match ! r with
      | Some(x) → x
      | None → assert(false)
    in complete ts turn; v
```

(a) Turn sequencer and single-element queue.

```
queueMPMC q = Λ.
    let slots = arrayInit q newSEQ in
    let pushTicket = ref(0) in
    let popTicket = ref(0) in
    (λv. enqueue slots q pushTicket v,
     λx. dequeue slots q popTicket)
enqueue slots q pushTicket v =
    let t = FAA(pushTicket, 1) in
    let idx = t mod q in
    let ticket = t/q in
    enqueueSEQ slots[idx] ticket v
dequeue slots q popTicket v =
    let t = FAA(popTicket, 1) in
    let idx = t mod q in
    let ticket = t/q in
    dequeueSEQ slots[idx] ticket v
```

(b) MPMC queue.

```
queueCG = Λ.
    let w = (newlock (), ref([])) in
    (λv. enqueueCG w v,
     λx. dequeueCG w)
enqueueCG (lk, hd) v =
    let rec go v ls =
      match ls with
      | [] → [v]
      | h :: t → h :: go v t
    in acquire lk;
    hd ← go v ! hd;
    release lk
dequeueCG (lk, hd) =
    acquire lk;
    match ! hd with
    | [] → assert(false)
    | h :: t → hd ← t;
               release lk;
               h
```

(c) Coarse-grained queue.

Figure 1. Implementations of the various algorithms.

number has been carried out. For an enqueue and a dequeue operations with the same turns, the enqueue is carried out first.

The single-element queue is implemented as a reference to an option type, protected by a *single* turn sequencer. In order to ensure that the turn sequencer operations are called with correct turns, the implementations of the enqueue and dequeue operations adhere to the following discipline. The even turns of the turn sequencer correspond to the enqueue operations and the odd turns correspond to the dequeue operations. Hence when enqueue_{SEQ} (dequeue_{SEQ}, respectively) is called with turn n, the corresponding turn for the turn sequencer is $2n$ ($2n + 1$, respectively). Not only does this allow for a single turn sequencer to provide turns for both of the operations, it also ensures that the enqueue and dequeue operations are carried out in the correct order. The first enqueue gets the first even turn, 0, the first dequeue gets the first odd turn, 1, and so on. Hence the enqueue and dequeue operations alternately get access to the pointer, and the dequeue operation can be sure that a value is present when it reads the pointer.

2.3 MPMC queue

The MPMC queue is a blocking queue of a fixed *capacity* q. The implementation of the MPMC queue is shown in Figure 1b. The binary operator "mod" denotes modulo (or remainder) and "/" denotes *integer* division (*i.e.,* $3/2 = 1$).

Upon initialization, an array of length q is created, with each entry containing a single-element queue. The function arrayInit constructs an array of the given length, calls the given function once for each entry and sets the entry to the result. In addition to the array, the queue contains two *ticket dispensers* (references to natural numbers): *pushTicket* and *popTicket*. The first keeps track of the tickets for the enqueue operation, and the second does the same for the dequeue operation.

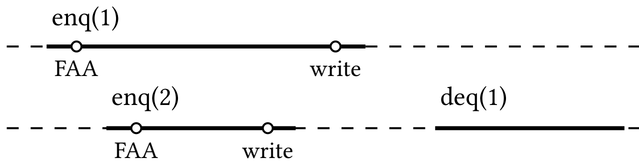
The enqueue operation first takes a ticket by incrementing the value of *pushTicket* with, FAA, which atomically increments the ticket and leaves enqueue with a ticket t. This ticket gives an index, $t \bmod q$, in the array for a single-element queue. Then, enqueue writes an element into the single-element queue by using the turn $\lfloor t/q \rfloor$. The dequeue operation proceeds in a similar way. It atomically increments *popTicket* and calculates an index and a turn in the same way. It dequeues a value from the single-element queue and returns this value.

3 Linearizability of the Folly MPMC queue

In this section we analyse the MPMC queue informally and identify its linearization points.

As a first guess, one might think that the linearization point for enqueue is when enqueue writes its value into the single-element queue and, similarly, for dequeue when it reads the value from the single-element queue. After all,

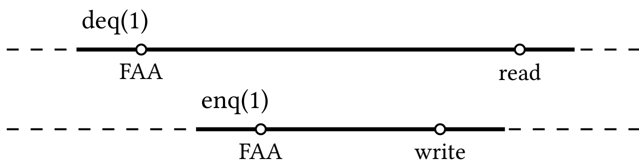
these are the points where a value is physically inserted into or read from the data structure. However, placing the linearization points in this way does not work, as the following example shows:



This diagram represents two threads executing operations on the queue. The filled segments represent the duration of the operations. In the example, the first enqueue executes its FAA and gets ticket 0. Afterwards the second enqueue executes its FAA, gets ticket 1, and writes its value to the queue. Then the first enqueue writes its value. Finally a dequeue executes; gets ticket 0, and therefore returns 1. To make this consistent, the linearization point of the first enqueue should happen before the linearization point of the second enqueue. But, the second enqueue writes its value into the queue *before* the first enqueue does so. Hence, making the linearization points at that time in enqueue is too late.

As the example suggests, the linearization point of the enqueue operations happens at the FAA. If an enqueue operation receives a ticket i, then clearly the value that it insert into the queue is eventually read and returned by the dequeue operation that also receives the ticket i. This means that exactly when the FAA in enqueue is executed, it is determined where in the queue its value is inserted. It thus makes senses to place the linearization point at the FAA. Following this line of argument, we say that the enqueue that receives ticket i is the ith enqueue. Moreover, we call the dequeue that receives ticket i the ith dequeue, and we say that the ith enqueue and the ith dequeue *correspond* to each other.

It might seem that the linearization point in dequeue is similarly at the FAA operation. This, however, does not always work, as the following example shows:



The crux of the example is that dequeue receives ticket 0 before the corresponding enqueue takes its ticket. It is therefore not consistent to put the linearization point of dequeue at its FAA, as dequeue would then take place before the value it returns is enqueued in the first place. However, in general, one can not place the linearization point at when dequeue reads the value either, as that would lead to the same problems as for enqueue.

Thus, the linearization point of the dequeue operation is not always fixed. Looking at the example, we see that we could place the linearization point for the waiting dequeue *just after* the linearization point for the enqueue that unblocks it. This means that the linearization point of dequeue happens during the execution of enqueue — an external linearization point.

In summary, we conclude the following. If the ith dequeue arrives *after* its corresponding enqueue then it has a fixed linearization point at its FAA. If, on the other hand, it arrives *before* its corresponding enqueue then it has an external linearization point, which happens right after the corresponding enqueue's linearization point. Observe that even with the external linearization point, it is the case that the ith dequeue always has its linearization point before the $(i + 1)$'th dequeue.

Abstract state. Given the placement of linearization points as above, we can talk about the *abstract state* of the queue, which is determined by the linearized order of the operations. Note that as soon as enqueue receives a ticket, its value becomes part of the abstract state, before it is even written into the array. Symmetrically, when a dequeue receives a ticket, it dequeues a value from the logical queue, even though that value is still present in the physical queue. Thus, the physical state of the underlying array does not determine the abstract state of the queue, *e.g.,* the queue might physically contain no values, while logically it contains arbitrarily many values (and vice versa).

Calculating the abstract state of the queue is important in the refinement proof (Sections 5 and 6), but it can not rely on the physical state of the array. The abstract is however directly related to the physical state of *pushTicket* and *popTicket*. If $popTicket \leq pushTicket$, then there are exactly $pushTicket - popTicket$ elements in the logical queue. Otherwise the queue is empty and there are $popTicket - pushTicket$ dequeue operations that have arrived before their corresponding enqueue. We will see how these considerations are formalized as part of the refinement proof in Section 6.

4 Specifications for the Turn Sequencer and the Single-Element Queue

In this section we give unary hoare-triple specifications for the turn sequencer and the single-element queue. We also sketch how these can be proved. We emphasize that the proof of the single-element queue only uses the *specification* (and not the implementation) of the turn sequencer. Similarly, when we prove contextual refinement for the MPMC queue, we only make use of the specification for the single-element queue. Thus our specifications and proofs are *modular*, and we observe that to prove contextual refinement for the MPMC queue, a unary specification for the single-element queue suffices.

4.1 Turn Sequencer

As mentioned earlier, the turn sequencer is a mechanism for mutual exclusion. Therefore, our specification of the turn

Turn Sequencer

$\{R(0)\}$ newTS () $\{v.\ \exists\gamma.\ \mathsf{isTS}(\gamma, R, v) * \mathsf{turns}^{\gamma}(\mathbb{N})\}$

$\{\mathsf{isTS}(\gamma, R, v) * \mathsf{turn}^{\gamma}(n)\}$ wait v n $\{R(n) * \mathsf{close}(v, n)\}$

$\{\mathsf{isTS}(\gamma, R, v) * R(n+1) * \mathsf{close}(v, n)\}$ complete v n $\{\mathsf{True}\}$

Single Element Queue

$\{\mathsf{True}\}$ new_{SEQ} () $\{v.\exists\gamma.\ \mathsf{isSEQ}(\gamma, Q, v) * \mathsf{turns}_e(\gamma, 0) * \mathsf{turns}_d(\gamma, 0)\}$

$\{\mathsf{isSEQ}(\gamma, Q, v) * \mathsf{turn}_e(\gamma, n) * Q(n, x)\}$ $\mathsf{enqueue}_{SEQ}$ v n x $\{\mathsf{True}\}$

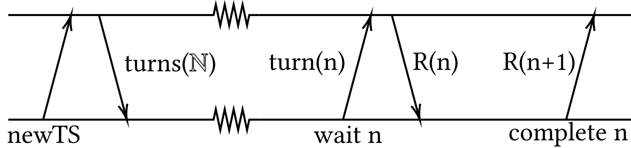
$\{\mathsf{isSEQ}(\gamma, Q, v) * \mathsf{turn}_d(\gamma, n)\}$ $\mathsf{dequeue}_{SEQ}$ v n $\{x.Q(n, x)\}$

Figure 2. Unary specifications for turn sequencer and single-element queue.

sequencer (shown in Figure 2) is an extension of a typical concurrent separation logic specification for a lock. There are two key differences though. The first difference is to enforce that clients of the turn sequencer uses turns correctly. For instance, wait should never be invoked with a past turn. The second difference is that the resource that the turn sequencer protects is indexed with a turn number.

The specification uses two predicates close and isTS, these are existentially quantified so that they are abstract to clients of the specification [1, 23]. The latter, isTS, is the representation predicate which is *persistent*; this intuitively means that, unlike other separation logic propositions, isTS is freely duplicable and not consumed by preconditions (the precise definition is given in Section 5).

The predicate R describes the resource that the turn sequencer protects. Whereas a lock protects a single resource $R : iProp$, the turn sequencer protects a $\mathbb{N}$-indexed *family* of resources, that is, $R : \mathbb{N} \rightarrow iProp$, where index represents the current turn. This generalization of the resource is possible since the turn sequencer guarantees to run clients in the order of their turns. When it becomes a clients turn to enter its critical section, it can rely on all earlier turns having been carried out. This allows for "threading" the resource through all the clients, as depicted in the diagram below where the turn sequencer is at the top and its clients at the bottom.



The $R(0)$ in the precondition of newTS ensures that when a turn sequencer is created, the turn sequencer owns the resource for the initial turn. When wait is called with turn n, the client receives the resource for that turn, $R(n)$. When completing the turn, the client must give back $R(n+1)$ and not $R(n)$. This makes it possible for the turn sequencer to give $R(n+1)$ to next thread in line (which is waiting for the turn $n+1$).

TURN-ALLOC
$$\frac{X \subseteq \mathbb{N}}{\Rrightarrow \exists\gamma.\ \mathsf{turns}^{\gamma}(X)}$$

TURN-SEP
$$\frac{X \cap Y = \emptyset}{\mathsf{turns}^{\gamma}(X) * \mathsf{turns}^{\gamma}(Y) \dashv\vdash \mathsf{turns}^{\gamma}(X \cup Y)}$$

TURN-DISJ
$$\frac{X \cap Y \neq \emptyset \qquad \mathsf{turns}^{\gamma}(X) \qquad \mathsf{turns}^{\gamma}(Y)}{\mathsf{False}}$$

Figure 3. Rules for turns.

We now consider the handling of turns in the specification. To represent turns we use *ghost state*, an Iris feature also found in other separation logics [4, 16, 22]. Ghost state are resources that do not correspond to any physical state of the program. In our case, we want a resource representing ownership over turns—where owning the turn n implies that one has the "right" to wait for the nth turn. The details of how this ghost state is defined are not important. The resource $\mathsf{turns}^{\gamma}(X)$ denotes ownership over the set of turns $X \subseteq \mathbb{N}$, and the singular $\mathsf{turn}^{\gamma}(n) \triangleq \mathsf{turns}^{\gamma}(\{n\})$ denotes ownership over a turn $n \in \mathbb{N}$. These turns can be manipulated, for instance by a client of the turn sequencer, using the rules in Figure 3. The *update modality*, $\Rrightarrow$, in these rules represents the possibility of updating ghost state and can safely be ignored. The rule TOKENS-ALLOC states that for any set of natural numbers one can construct a resource for them with a fresh *ghost name* γ. The ghost name can be thought of as a location or variable for the ghost state. Ownership over two disjoint sets of turns, combines into ownership over their union (TURN-SEP). Ownership over two sets of turns that are *not* disjoint is a contradiction (TURN-DISJ). This matches the intuition that two turns can not both have the right to wait for the same turn.

As depicted in the diagram above, when a client creates a new turn sequencer, it acquires ownership over *all* turns: $\mathsf{turns}^{\gamma}(\mathbb{N})$. Notice how this requires that we can represent an *infinite* set of turns as ghost state. For readers familiar with Iris, we note that we could instead have used the common approach of using an authorative resource algebra of *finite* sets. This would however have made the specification slightly more cumbersome to use as one would then have to allocate new turns by updating the authorative element. The benefits of our approach are even more evident when we use it to show the specification for the single-element queue. We refer to our Coq formalization for the technical details on how we encode infinite sets using resource algebras.

To call wait for a turn n the client must own $\mathsf{turn}^{\gamma}(n)$, the ownership of which is then transferred into the turn sequencer, ensuring that the client can only wait for the same turn once.

Finally, when a client acquires the current turn, it gets $\mathsf{close}(v, n)$, an exclusive resource giving permission to complete the turn.

Proof of Specification (Sketch). To prove that the implementation of the turn sequencer meets the specification, we use the following definitions of the predicates:

$$\mathrm{close}(\ell, n) \triangleq \ell \overset{1/2}{\hookrightarrow} n$$

$$\mathrm{isTS}(\gamma, R, \ell) \triangleq \boxed{\exists n.\, \ell \overset{1/2}{\hookrightarrow} n * \mathrm{turns}^{\gamma}(\{m \in \mathbb{N} \mid m < n\}) * (R(n) * \mathrm{close}(\ell, n) \vee \mathrm{turn}^{\gamma}(n))}^{\mathcal{N}}$$

For newTS, we have the resource $R(0)$ from the precondition and we obtain $\ell \hookrightarrow 0$ from stepping through the implementation. We can then allocate the ghost state $\mathrm{turns}^{\gamma}(\mathbb{N})$ using TURN-ALLOC. This allows us to establish the invariant by picking the left disjunct therein.

For wait, we open the invariant around the load. We then have the points-to predicate for the location, and can consider whether the value stored in the location is equal to the turn that wait was called with. In the latter case, we can use induction to handle the recursive call when the check in if fails. In the former case, the $\mathrm{turn}^{\gamma}(n)$ in the right disjunct in the invariant leads to a contradiction, due to the $\mathrm{turn}^{\gamma}(n)$ in the precondition. We thus have the resources in the left disjunct which we can use to show the postcondition, and then close the invariant by showing the right disjunct.

Finally, for complete, we use $\mathrm{close}(\ell, n)$ in the precondition to conclude that n is still the current turn, *i.e.,* the existential is equal to n. This is the case since $\mathrm{close}(\ell, n)$ is in fact half of the points-to predicate for ℓ. We then have a contradiction in the right disjunct in the disjunction, and symmetrically to what we did for wait, we "flip" the disjunction when we close the invariant.

4.2 Single-Element Queue

Similarly to the specification for the turn sequencer, in the specification for the single-element queue (shown in Figure 2) we must ensure that no two dequeue or enqueue operations are performed with the same turn. As such, creating a new single-element queue gives ownership over two sets of turns: one for enqueue and another one for dequeue. These, $\mathrm{turns}_e(\gamma, n)$ and $\mathrm{turns}_d(\gamma, n)$, denote ownership over all the turns for enqueue and dequeue, respectively, except for the first n such turns. Additionally, $\mathrm{turn}_e(\gamma, n)$ and $\mathrm{turn}_d(\gamma, n)$ represent ownership over the nth turn for enqueue and dequeue, respectively. When calling $\mathrm{enqueue}_{SEQ}$ or $\mathrm{dequeue}_{SEQ}$ with n, the specification requires the corresponding turn.

The representation predicate isSEQ is parameterized by a predicate $Q : \mathbb{N} \to v \to \mathit{iProp}$. If x is the nth value added to the queue, then $Q(n, x)$ should hold. Correspondingly, the specification for $\mathrm{enqueue}_{SEQ}$ requires this in its precondition. This in turn allows the specification for $\mathrm{dequeue}_{SEQ}$ to ensure, in its postcondition, that the returned value satisfies the predicate.

Proof of Specification (Sketch). To prove that the implementation of the single-element queue meets the specification, we first consider the definition of turn_e and turn_d. These are defined to be ownership over all the *even* and the *odd* turns, respectively, except for the first n even or odd numbers:

$$\mathrm{turns}_e(\gamma, n) \triangleq \mathrm{turns}^{\gamma}(\{m \in \mathbb{N} \mid \mathrm{even}(m) \wedge 2n \leq m\})$$
$$\mathrm{turns}_d(\gamma, n) \triangleq \mathrm{turns}^{\gamma}(\{m \in \mathbb{N} \mid \mathrm{odd}(m) \wedge 2n + 1 \leq m\})$$
$$\mathrm{turn}_e(\gamma, n) \triangleq \mathrm{turn}^{\gamma}(2n) \qquad \mathrm{turn}_d(\gamma, n) \triangleq \mathrm{turn}^{\gamma}(2n + 1)$$

Notice how these definitions are only possible because the specification for the underlying turn sequencer allows for ownership over *infinite* sets of turns.

Next we define the representation predicate isSEQ. It states that the value making up the single-element queue is a pair of a location and a turn sequencer. The representation predicate for the underlying turn sequencer is instantiated with the resource R_{SEQ}, which states that if the current turn is even, then the location points to **None**, and otherwise it points to a **Some** v. Since the n given to R_{SEQ} is a turn for the turn sequencer, we must convert it to get a turn for the single-element queue. This is why R_{SEQ} applies Q to $(n-1)/2$.

$$R_{\mathrm{SEQ}}(Q, \ell)(n) \triangleq \begin{cases} \ell \hookrightarrow \textbf{None} & \text{if even}(n) \\ \exists v.\, \ell \hookrightarrow \textbf{Some}\ v * Q((n-1)/2, v) & \text{otherwise} \end{cases}$$

$$\mathrm{isSEQ}(\gamma, Q, v) \triangleq \exists ts, \ell.\, v = (ts, \ell) * \mathrm{isTS}(\gamma, R_{\mathrm{SEQ}}(Q, \ell), ts)$$

We can prove the specification for new_{SEQ} by appealing to the specification for the turn sequencer, and then splitting the set of all turns into the set of all the even turns and set of all the odd turns. The specification for $\mathrm{enqueue}_{SEQ}$ and $\mathrm{dequeue}_{SEQ}$ are proved simply by using the specification for the turn sequencer. When $\mathrm{enqueue}_{SEQ}$ for the nth enqueue completes its turn, the turn for the turn sequencer is $2n$, meaning that it must show R_{SEQ} for $2n + 1$, which requires it to show Q for n, which is in the precondition. Similar arithmetic show that when $\mathrm{enqueue}_{SEQ}$ for n gets its turn, it receives Q for n, which it can keep to show the precondition since it only has to show $\ell \hookrightarrow$ **None** when it completes its turn.

5 Proof of Contextual Refinement

As mentioned, our main result is that the MPMC queue is a contextual refinement of a coarse-grained queue.

Theorem 5.1. *For all $q \in \mathbb{N}$ where $q > 0$,*

$$\vdash \mathit{queue}_{MPMC}\ q \precsim_{ctx} \mathit{queue}_{CG} : \forall \alpha.(1 \to \alpha) \times (\alpha \to 1).$$

To prove this, we show the following *refinement judgment*:

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

This reads: the MPMC queue (the implementation) is a contextual refinement of the coarse-grained queue (the specification). The *soundness theorem* of ReLoC then states that if the

REL-PAIR
$$\frac{\Delta \models e_1 \precsim t_1 : \tau_1 \qquad \Delta \models e_2 \precsim t_2 : \tau_2}{\Delta \models (e_1, e_2) \precsim (t_1, t_2) : \tau_1 \times \tau_2}$$

REL-TLAM
$$\frac{\forall \tau_i : Val \times Val \to iProp.\ \square\left([\alpha := \tau_i], \Delta \models e_1 \precsim e_2 : \tau\right)}{\Delta \models \Lambda.e_1 \precsim \Lambda.e_2 : \forall \alpha.\tau}$$

REL-INV-ALLOC
$$\frac{P \qquad \boxed{P}^{\mathcal{N}} \twoheadrightarrow \Delta \models e_1 \precsim e_2 : \tau}{\Delta \models e_1 \precsim e_2 : \tau}$$

REL-LAM
$$\frac{\square\left(\forall v_1, v_2.\ [\![\tau]\!]_\Delta(v_1, v_2) \twoheadrightarrow \Delta \models (\lambda x_1.\ e_1)\ v_1 \precsim (\lambda x_2.\ e_2)\ v_2 : \sigma\right)}{\Delta \models (\lambda x_1.\ e_1) \precsim (\lambda x_2.\ e_2) : \tau \to \sigma}$$

REL-HOARE
$$\frac{\{P\}\ e_1\ \{Q\} \qquad P \qquad \forall v.\ (Q(v) \twoheadrightarrow \Delta \models K[\,v\,] \precsim e_2 : \tau)}{\Delta \models K[\,e_1\,] \precsim e_2 : \tau}$$

REL-PURE-L
$$\frac{e_1 \overset{\text{pure}}{\rightsquigarrow} e_1' \qquad \triangleright(\Delta \models K[\,e_1'\,] \precsim e_2 : \tau)}{\Delta \models K[\,e_1\,] \precsim e_2 : \tau}$$

REL-ALLOC-L
$$\frac{\forall \ell.\ \ell \hookrightarrow v \twoheadrightarrow \Delta \models K[\,\ell\,] \precsim e_2 : \tau}{\Delta \models K[\,\mathsf{ref}(v)\,] \precsim e_2 : \tau}$$

REL-LOAD-L
$$\frac{\ell \hookrightarrow v \qquad \ell \hookrightarrow v \twoheadrightarrow \Delta \models K[\,v\,] \precsim e_2 : \tau}{\Delta \models K[\,!\,\ell\,] \precsim e_2 : \tau}$$

Figure 4. Basic ReLoC rules (selection).

REL-QUEUE-R
$$\frac{\forall w.\ \mathrm{I}_{\mathrm{CG}}(w, \vec{x}) \twoheadrightarrow\ \models t \precsim K[\,w\,] : \tau}{\models t \precsim K[\,(\mathsf{newlock}\ (), \mathsf{ref}([\,]))\,] : \tau}$$

REL-ENQUEUE-R
$$\frac{\mathrm{I}_{\mathrm{CG}}(w, \vec{x}) \qquad (\mathrm{I}_{\mathrm{CG}}(w, \vec{x} +\!\!+ [v]) \twoheadrightarrow\ \models t \precsim K[\,()\,] : \tau)}{\models t \precsim K[\,\mathsf{enqueue}_{\mathrm{CG}}\ w\ v\,] : \tau}$$

REL-DEQUEUE-R
$$\frac{\mathrm{I}_{\mathrm{CG}}(w, v :: \vec{x}) \qquad (\mathrm{I}_{\mathrm{CG}}(w, \vec{x}) \twoheadrightarrow\ \models t \precsim K[\,v\,] : \tau)}{\models t \precsim K[\,\mathsf{dequeue}_{\mathrm{CG}}\ w\,] : \tau}$$

REL-DEQUEUE-DETACHED
$$\frac{\mathrm{I}_{\mathrm{CG}}(w, v :: \vec{x}) \qquad (\models - \precsim_{id} \mathsf{dequeue}_{\mathrm{CG}}\ w : -)}{\Rrightarrow \mathrm{I}_{\mathrm{CG}}(w, \vec{x}) * (\models - \precsim_{id} v : -)}$$

Figure 5. Symbolic execution rules for the coarse-grained queue.

above refinement judgment is provable in ReLoC, then the contextual refinement is true in the meta-logic (*e.g.,* Coq or math). The advantage to this approach is that ReLoC offers high-level rules for deriving the refinement judgment. A selection of the rules are given in Figure 4, these are explained as we go along.

As the first step of the proof we observe that the MPMC queue and the coarse-grained queue both consists of a type abstraction (*i.e.,* they are of the form $\Lambda.e$) corresponding to he universally quantified type α. Hence we can use the *structural rule* REL-TLAM. Structural rules use the types and apply when both the implementation and the specification are of the same shape. Specifically, REL-TLAM states that we must assume that the type α corresponds to a relation on values τ_i. Intuitively τ_i relates element in the MPMC queue with elements in the coarse-grained queue, it could for instance specify what it means for pairs to be related if τ is a pair type.

Under this assumption we must show that the bodies under the lambdas are related. Since these are not of the same shape, we use *symbolic execution rules*. These makes it possible to step forward the expression on either side. In the initialization of the MPMC queue we, among other such rules, use REL-ALLOC-L to step over the allocation of the references. At the pair in $\mathsf{queue}_{\mathrm{CG}}$ we apply the derived rule REL-QUEUE-R and obtain the representation predicate $\mathrm{I}_{\mathrm{CG}}(w, [\,])$. After the symbolic execution we have the resources

$$\begin{aligned} &\ell_{pop} \hookrightarrow 0 * \ell_{push} \hookrightarrow 0 * \\ &\ell_{arr} \hookrightarrow_* \text{map } \pi_2\ SEQs * \mathrm{I}_{\mathrm{CG}}(w, [\,]) \end{aligned} \tag{1}$$

where *SEQs* is a list of pairs of ghost names and values. Since arrayInit calls new_{SEQ} q times and we apply the specification for new_{SEQ} for each of them we get q values and ghost names, these are what we have in *SEQs*. The definition of I_{CG} can be considered abstract, we only need the rules in Figure 5 (the definition can be found in our Coq formalization). When using the specification for new_{SEQ} we also get the representation predicate for each single-element queue and must decide on a Q, we describe this choice in the next section.

With the resources described above we now have to show:

$$\begin{aligned} [\alpha := \tau_i] \models\ &(\lambda v.\ \mathsf{enqueue}\ \ell_{arr}\ q\ \ell_{push}\ v, \lambda x.\ \mathsf{dequeue}\ \ell_{arr}\ q\ \ell_{pop}) \\ &\precsim (\lambda v.\ \mathsf{enqueue}_{\mathrm{CG}}\ w\ v, \lambda x.\ \mathsf{dequeue}_{\mathrm{CG}}\ w) \\ &:\ (1 \to \alpha) \times (\alpha \to 1). \end{aligned}$$

At this point we apply the structural rule REL-PAIR which requires us two show that the two functions refine the coarse-grained counterparst. To show that the functions are related, we use REL-LAM and must show that they are related on related arguments. The precondition of that rule is guarded by the *persistence modality* $\square$ from Iris. Intuitively, $\square P$ states that P always holds, and to prove it one must show P without relying on any non-persistent resources. A proposition P is said to be *persistent*, if one can derive $\square P$ from P. In REL-LAM the persistence modality, means we have to show that the applied functions are *always related*. This reflects that for two functions to be related they have to be indistinguishable in any context – including a context that calls the functions several times, potentially in parallel.

This means that the resources in Equation (1) can not be used directly. Instead we must use them to establish an *invariant*. An invariant $\boxed{P}^{\mathcal{N}}$ persistently states that the proposition P always holds ($\mathcal{N}$ is the name of the invariant). A invariant can be created using REL-INV-ALLOC. The contents of the invariant can be accessed for the duration of a single execution step of the implementation (we call this *opening an invariant*). After taking the step, the user has to show that the invariant still holds (we call this *closing an invariant*). Before closing the invariant, the user can symbolically execute the specification. The number of execution steps for the specification is not restricted; intuitively, this is because we are showing a form of *simulation*, matching every execution step in the specification with (possibly) multiple execution steps in the specification[2].

Thus, we must now use the resources we have to establish an invariant that relates the abstract state of the MPMC queue with that of the coarse-grained queue. Defining a suitable invariant is the most challenging part of the refinement proof, and we explain it in detail in the next section. We finish the proof in Section 8, where we use the invariant to show that enqueue and dequeue refine their coarse-grained counterparts.

6 Invariant for Refinement Proof

The invariant we use is shown in Figure 6. It is non-trivial and is key to the refinement proof so we devote this section to explain its different parts. Overall, the invariant keeps track of the physical state of the queue, ensures that the queue represents a logic-level list of values, manages the turns for all the single-element queues, and handles the external linearization point.

The invariant is parameterized by the interpretation of the type of values stored in the queue (τ_i), ghost names (γ_t, γ_m, γ_l), the size of the queue (q), the values for the MPMC queue (ℓ_{pop}, ℓ_{push}, ℓ_{arr}, $SEQs$) and the value for the coarse-grained queue (w).

Notice the headers in the presentation of the invariant, we cover each of these parts of the invariant in turn.

Relation to coarse-grained queue. The invariant states the existence of two lists xs_i and xs_s. The state of the coarse-grained queue is tied to xs_s by $\mathrm{I_{CG}}(w, xs_\mathrm{s})$ and xs_i is tied to the MPMC queue by the rest of the invariant. The separating conjunction over the two lists thus states that the abstract states of the two queues are relate at type τ_i.

Physical State. The physical state of the queue is rather simple. The queue consists of three locations and the invariant contains points-to predicates for all three. As the pointer to the array never changes we represent it using the persistent points-to predicate $\hookrightarrow^{\square}_{*}$ [31].

[2]The way that the condition on the number of steps is enforced is rather technical; see [9] for the details.

Ghost List. We previously explained how the physical state of the queue reveals very little about the actual values stored in the queue. We use a *ghost list* to remedy this.

The ghost list contains *all* values that have been enqueued, in particular this includes values that are no longer present in the queue. Thus, while the physical state does not change when enqueue executes its FAA, the ghost state does. And, since the linearization point of enqueue is when it increments *pushTicket*, the number of values that have been added to the queue is always exactly *pushTicket*. Hence, the ghost list is connected with the physical state in part from the requirement that its length is equal to the value of *pushTicket*.

Ownership of a ghost list corresponding to the logical list xs is denoted $\mathsf{list}^{\gamma_l}(xs)$. The resource for the ghost list is ephemeral as it changes over time, when new values are enqueued at the end. This is in fact the *only* way in which the ghost list changes. This makes it possible to include a *persistent* resource, $\mathsf{listAt}^{\gamma_l}(i, x)$, denoting the knowledge that the ith value added to the queue is x. Indeed, the reason for using the ghost list is exactly to be able to obtain this persistent predicate. Had the ghost list instead reflected only the current list of values in the queue, we would not have been able to make any persistent statements about it, as it could change entirely over time. The ghost list satisfies a number of proof rules presented in Figure 7; these rules are sufficient to carry out the proof.

We can now move on to the following part of the invariant:

$$\mathsf{list}^{\gamma_l}(m) * |m| = pushTicket * \mathsf{drop}(popTicket, m) = xs_\mathrm{i}$$

As alluded to above, the ghost list corresponds to a logical list m and the length of m is *pushTicket*. Moreover, if we remove the first *popTicket* elements from m (denoted $\mathsf{drop}(popTicket, m)$), then the remaining list is equal to xs_i— the list which represents the abstract state of the queue. This makes sense, since the ghost list contains all values that have been enqueued and we remove exactly those that have also been dequeued. Note that when $pushTicket \leq popTicket$, then the above implies that xs_i is empty.

Invariants for Single-Element Queues. For each of the q single-element queues in the array the invariant needs to include the invariant for the single-element queue and to manage its turns.

We need to instantiate the invariant for each single-element queue with the predicate that holds for the values in it. Recall that this resource is parameterized by the value in the queue and its index. We define Q to state that the value in the single-element queue corresponds to the "right" value in the ghost list:

$$Q(i)(j, v) \triangleq \mathsf{listAt}^{\gamma_l}(jq + i, v)$$

Here q is the capacity of the queue and i is the index of the particular single-element queue. For the jth element added to this single-element queue we can then calculate the position of this element in the whole queue as $jq + i$. Thus, going

881 882 883 884 885 886 887 888 889 890 891 892 893 894 895 896 897 898 899 900 901 902 903 904 905 906 907 908 909 910 911 912 913 914 915 916 917 918 919 920 921 922 923 924 925 926 927 928 929 930 931 932 933 934 935

$$I(\tau_i, \gamma_t, \gamma_m, \gamma_l, q, \ell_{pop}, \ell_{push}, \ell_{arr}, SEQs, w) \triangleq \exists xs_i, xs_s, popTicket, pushTicket, m.$$

$$\overbrace{\begin{array}{c}\ell_{pop} \hookrightarrow popTicket * \ell_{push} \hookrightarrow pushTicket * \\ \ell_{arr} \hookrightarrow_*^{\square} \text{map } \pi_2\ SEQs\end{array}}^{\textbf{Physical state}} * \overbrace{\begin{array}{c}\text{list}^{\gamma_l}(m) * |m| = pushTicket * \\ \text{drop}(popTicket, m) = xs_i\end{array}}^{\textbf{Ghost list}} * \overbrace{|SEQs| = q * \left(\mathop{\bigstar}_{i=0}^{q} I_{SEQ}(i, SEQs_i)\right)}^{\textbf{Invariants for the single-element queues}}$$

$$\underbrace{\begin{array}{c}\text{tokensFrom}^{\gamma_t}(\max(popTicket, pushTicket)) * \text{ids}^{\gamma_m}(popTicket) * \\ \left(\mathop{\bigstar}_{i=0}^{pushTicket-1} \text{pushObl}(i)\right) * \left(\mathop{\bigstar}_{i=pushTicket}^{popTicket-1} \exists id. \begin{array}{l}\text{idsAt}^{\gamma_m}(i, id) * \\ \models - \precsim_{id} \text{dequeue}_{CG}\ w : -\end{array}\right)\end{array}}_{\textbf{Handling of external linearization point}} * \underbrace{\text{I}_{CG}(w, xs_s) * \mathop{\bigstar}_{(x_i, x_s) \in (xs_i, xs_s)} \tau_i(x_i, x_s)}_{\textbf{Relation to coarse-grained queue}}$$

where

$$\text{pushObl}(i) \triangleq \text{token}^{\gamma_t}(i) \vee (\exists id, v_i, v_s.\ \text{idsAt}^{\gamma_m}(i, id) * (\models - \precsim_{id} v_s : -) * \tau_i(v_i, v_s) * \text{listAt}^{\gamma_l}(i, v))$$

$$I_{SEQ}(i, (\gamma, v)) \triangleq \text{isSEQ}(\gamma, Q(i), v) * \text{turnCtx}(\gamma, i)$$

$$\text{turnCtx}(\gamma, i) \triangleq \text{turns}_e(\gamma, \text{affectingOps}(pushTicket, q)) * \text{turns}_d(\gamma, \text{affectingOps}(popTicket, q))$$

$$\text{affectingOps}(ops, q) \triangleq \lfloor ops/q \rfloor + (\text{if } (i < ops \bmod q) \text{ then } 1 \text{ else } 0)$$

$$Q(i)(j, v) \triangleq \text{listAt}^{\gamma_l}(jq + i, v)$$

Figure 6. Invariant for the MPMC queue

GHOST-LIST-ALLOC
$$\frac{}{\Rrightarrow \exists \gamma_l.\ \text{list}^{\gamma_l}([])}$$

GHOST-LIST-APPEND
$$\frac{\text{list}^{\gamma_l}(xs)}{\Rrightarrow \text{list}^{\gamma_l}(xs +\!\!+ [x]) * \text{listAt}^{\gamma_l}(|xs|, x)}$$

GHOST-LIST-AGREE
$$\frac{\text{listAt}^{\gamma_l}(i, x) \qquad \text{listAt}^{\gamma_l}(i, x')}{x = x'}$$

GHOST-LIST-LOOKUP
$$\frac{\text{list}^{\gamma_l}(xs) \qquad xs_i = x}{\Rrightarrow \text{list}^{\gamma_l}(xs) * \text{listAt}^{\gamma_l}(i, x)}$$

Figure 7. Rules for the ghost list.

TOKENS-ALLOC
$$\Rrightarrow \exists \gamma_t.\ \text{tokensFrom}^{\gamma_t}(0)$$

TOKEN-EXCLUSIVE
$$\frac{\text{token}^{\gamma_t}(n) \qquad \text{token}^{\gamma_t}(n)}{\text{False}}$$

TOKENS-TAKE
$$\frac{\text{tokensFrom}^{\gamma_t}(i)}{\text{tokensFrom}^{\gamma_t}(i+1) * \text{token}^{\gamma_t}(i)}$$

Figure 8. Rules for tokens.

through the ghost list, we connect the abstract state of the queue to the physical state of all the single-element queues.

In addition we must keep track of the turns for each single-element queue. We must calculate these turns based on the current value of *popTicket* and *pushTicket*. The affectingOps aids in this. Given the "global" count of an operation (dequeue or enqueue), it calculates how many times the single-element queue in question was affected.

Handling of External Linearization Point. The part of the invariant for handling the external linearization point is rather intricate. For the ith pair of operations, either enqueue or dequeue arrives first. In the former case the invariant should allow both enqueue and dequeue to carry out their own linearization point. In the latter case the invariant must facilitate handling of the external linearization point.

To do this we must intuitively encode the following: When dequeue opens the invariant around its FAA it must transfer the requisite resources into the invariant that will allow another thread to carry out its linearization point. Then, when enqueue opens the invariant around its FAA it should be forced to carry out the corresponding dequeue's linearization point and transfer the result into the invariant. Later, dequeue needs to open the invariant again, conclude that its linearization point has been carried out, and be able to transfer the resources for the executed linearization point out of the invariant.

Came-first token. To keep track of which operation came first we use a *tokens*—created using ghost state very similarly to what we did for turns earlier. The rules for tokens are seen in Figure 8. The ith dequeue or enqueue that comes first will be able to take the token $\text{token}^{\gamma_t}(i)$. Hence, owning $\text{token}^{\gamma_t}(i)$ proves that an operation came before its corresponding counterpart. The invariant owns all the tokens where neither operation has taken a ticket:

$$\text{tokensFrom}^{\gamma_t}(\max(popTicket, pushTicket)).$$

To see how this allows the operation that arrives first to take a ticket, note that when enqueue and dequeue open the invariant around their FAA, they will close the invariant by using

936 937 938 939 940 941 942 943 944 945 946 947 948 949 950 951 952 953 954 955 956 957 958 959 960 961 962 963 964 965 966 967 968 969 970 971 972 973 974 975 976 977 978 979 980 981 982 983 984 985 986 987 988 989 990

$$\frac{}{\Rrightarrow \exists \gamma_m.\ \mathsf{ids}^{\gamma_m}(0)}\ \textsc{identifier-alloc} \qquad \frac{\mathsf{ids}^{\gamma_m}(n)}{\Rrightarrow \mathsf{ids}^{\gamma_m}(n+1) * \mathsf{idsAt}^{\gamma_m}(n, id)}\ \textsc{identifier-decide}$$

$$\frac{\mathsf{ids}^{\gamma_m}(n)}{\mathsf{ids}^{\gamma_m}(n+1)}\ \textsc{identifier-skip} \qquad \frac{\mathsf{idsAt}^{\gamma_m}(i, id) \qquad \mathsf{idsAt}^{\gamma_m}(i, id')}{id = id'}\ \textsc{identifier-agree}$$

Figure 9. Rules for identifier registry.

$pushTicket + 1$ and $popTicket + 1$, respectively, for the existential variable that they introduced. If enqueue comes first then $popTicket \leq pushTicket$. Hence $\max(popTicket, pushTicket)$ is equal to $pushTicket$, and only $\mathsf{tokensFrom}^{\gamma_t}(pushTicket{+}1)$ is required for closing the invariant and one token can be kept by enqueue per the rule tokens-take. On the other hand, if enqueue is last, then $pushTicket < popTicket$ and

$$\max(popTicket, pushTicket) = \max(popTicket, pushTicket{+}1).$$

Thus when closing the invariant, all the tokens are required and none can be kept. For dequeue the situation is symmetric. All in all, this means that this construction ensures that ith operation that comes first can take the i'th token.

Identifier registry. Concretely for enqueue to carry out it corresponding dequeue's linearization point means that it should step dequeue's specification forward. To this end, $\models - \precsim_{id} e : -$ represents that some thread, identified by id, needs to show that its implementation refines e. This resource is part of the extensions that we make to ReLoC and is explained in greater detail in Section 7. For now it suffices to know that the state of dequeue's specification is associated with an identifier, id, and that dequeue needs a way to ensure that enqueue steps precisely the specification with that identifier forward. To support this, the invariant contains a resource that lets the ith dequeue *register* which identifier it has. The rules for this construction are shown in Figure 9. The resource $\mathsf{ids}^{\gamma_m}(n)$ represents that only the n first dequeue operations might have registered an identifier. The persistent resource $\mathsf{idsAt}^{\gamma_m}(i, id)$ represents the knowledge that the ith dequeue has registered the identifier id.

Pending dequeues. When $pushTicket < popTicket$, there are $popTicket - pushTicket$ dequeue operations blocked, waiting for a value to read. These blocked dequeues are exactly those with external linearization points, and when an enqueue comes along, it should carry out the corresponding dequeue's linearization point. To this end, enqueue needs some resources, which we store in the invariant:

$$\mathop{\Huge *}_{i=pushTicket}^{popTicket-1} \exists id.\ \mathsf{idsAt}^{\gamma_m}(i, id) * \left(\models - \precsim_{id} \mathsf{dequeue}_{\mathsf{CG}}\ w : - \right)$$

This reads: every ith dequeue operation, where $pushTicket \leq i < popTicket$, has stored some identifier in the identifier registry and we have the corresponding right refinement, which is ready to invoke dequeue on the coarse-grained queue.

Enqueue obligation. The final piece in the invariant is

$$\mathop{\Huge *}_{i=0}^{pushTicket-1} \mathsf{pushObl}(\gamma_l, \gamma_t, \gamma_m, i).$$

Since enqueue closes the invariant with $pushTicket{+}1$, it must show $\mathsf{pushObl}(\gamma_l, \gamma_t, \gamma_m, pushTicket)$. Hence, one should think of $\mathsf{pushObl}(i)$ as: something which enqueue is obliged to show when it takes the ith ticket. Since pushObl is a disjunction, there are two ways for enqueue to meet this obligation. When enqueue comes first, the obligation is trivial: it can take the token $\mathsf{token}^{\gamma_t}(pushTicket)$, and this is exactly the first disjunct. If, on the other hand, enqueue is last, then there is no way to show the first disjunct and the only option is to show the second disjunct. This involves showing all the things that dequeue needs, including handling dequeue's linearization point.

Establishing the Invariant. To proceed the proof we must now establish the invariant. We must allocate the ghost state using the rules ghost-list-alloc, tokens-alloc, and identifier-alloc. As the parameters for the invariant we choose pick those that correspond from Section 5. When showing the invariant for the existentials we of course pick empty lists for xs_{i} and xs_{s} and 0 for $popTicket$ and $pushTicket$.

7 Extending ReLoC with Support for External Linearization Points

When verifying refinement for a data structure with an external linearization point, the operation with the external linearization point must let another operation "carry out its linearization point". In the context of ReLoC, where we proof refinements by symbolically executing two expressions, this means that the other operation must symbolically execute the specification of the first operation. Moreover, in the proof of the first operation it must be possible to keep symbolically executing the implementation up to the point where it can be concluded that another thread has stepped its specification forward. To facilitate this, we should be able to split the state of the implementation and the specification into two separate resources, and transfer the specification side to another thread. This, however, is not possible with the proof rules of ReLoC, because the refinement judgment couples the implementation and specification together.

To support reasoning about external linearization points in ReLoC, we therefore have to extend the logic with additional rules that allows the user to temporarily decouple a refinement judgment, such that the specification part can be symbolically executed independently of the implementation

part. A selection of the new rules for external linearization points are shown in Figure 10.

The rule REL-SPLIT states that one can *split* a refinement judgment into a *left* refinement $\models e_1 \precsim_{id} - : \tau$ and a *right* refinement $\models - \precsim_{id} e_2 : -$, which represent the state of the implementation and the specification, respectively. One can think of the left refinement as a normal refinement judgment where the specification has been *detatched*. When we split a refinement judgment, we naturally want to keep track of the fact that the two split, left and right, refinements originate from the same refinement judgment. We therefore parameterize the split refinement judgments by an *identifier id*, from some opaque set Id of identifiers. The rule REL-COMBINE symmetrically allows one to combine two sides of a refinement into a normal refinement.

Note that in both rules, the right refinement ($\models - \precsim_{id} e_2 : -$) appears on the left-hand side of the magic wand $-\!\!*$, *e.g.*, REL-SPLIT states that one may *assume* the right hand side and should then prove the left hand side. This allows us to treat the refinement ($\models - \precsim_{id} e_2 : -$) as a resource and, *e.g.*, , transfer it between different threads, for instance by putting it inside an invariant as in the previous section.

In addition to these two rules, we need symbolic execution rules to step forward the left and right refinement judgments separately. For the left refinement we *generalize* all of ReLoCs existing symbolic execution rules to apply *both* when a specification is present and when it is detached. Hence, one can consider a rule such as REL-PURE-L to apply to both a normal refinement (where a specification is present) *and* a left refinement (where an identifier is present). This is particularly beneficial in the Coq formalization where we have generalized ReLoC's existing lemmas and tactics such that a user can keep applying the same lemmas and tactics even after having detached the specification by using REL-SPLIT.

By contrast, the symbolic execution rules for the detached right-hand side look different, compared to the standard ReLoC symbolic execution rules (for reasons of space, we only show one such rule, REL-RIGHT-LOAD). Typically, symbolic execution rules facilitate *backwards* reasoning. This is the case for the "standard" ReLoC rules (Figure 4) and rules for the left refinements. However, per the explanation above, the right refinement is a resource one introduces into the context where one want to do *forward* reasoning which is exactly what the rules support.

In order to implement those rules, we had to make a number of changes to the model of ReLoC (described in [9]). For the reasons of space, and the fact that understanding those changes requires familiarity with Iris and ReLoC, we omit the details about the model from the paper, and direct an interested reader to the Coq formalization.

8 Refinement of enqueue and dequeue

We now complete the proof of refinement by showing that each of the operations refine their coarse-grained counterpart while assuming the invariant. We have already explained many of the ideas of the proof in the context of the invariant. Here we simply sketch the overall structure of the proof, show how we use the specification for the single-element queue and, in particular, how the external linearization point is handled, including how we use the new and generalized rules covered in Section 7.

8.1 Proof of enqueue

For enqueue we must prove the following refinement:

$$[\alpha := \tau_i] \models \lambda v.\, \mathsf{enqueue}\ \ell_{arr}\ q\ \ell_{push}\ v \precsim \lambda v.\, \mathsf{enqueue}_{CG}\ w\ v : \alpha \to 1.$$

We use REL-LAM and assume v_i and v_s where $[\![\tau_i]\!]_\Delta(v_s, v_i)$. We symbolically execute the implementation up to the FAA and open the invariant around it. This is the critical part of the proof. From the invariant we obtain the points-to predicate $\ell_{push} \hookrightarrow pushTicket$, and as the value of ℓ_{push} is changed from $pushTicket$ to $pushTicket+1$ we must close the invariant with $pushTicket + 1$ as the witness for the existential variable $pushTicket$. To see what this requires we must look at all the occurrences of $pushTicket$ in the invariant.First of all, from the definition of turnCtx we can take out $\mathsf{turn}_e(\gamma, n)$ for the enqueue operation of the ($pushTicket$ mod q)'th single-element queue with $n = \lfloor pushTicket/q \rfloor$. Then, we have to update the ghost list. We append an element to the ghost list using GHOST-LIST-APPEND. This gives us the persistent $\mathsf{listAt}^{\gamma_l}(pushTicket, v_i)$ and a ghost list that is one element longer. We now carry out enqueue's linearization point, meaning we symbolically execute the specification side by using REL-ENQUEUE-R. Afterwards the abstract state of the coarse-grained queue is $xs_s + [v_s]$. If $popTicket \leq pushTicket$ we can now trivially close the invariant per the discussion in Section 6. Otherwise we must handle the external linearization point by showing the right disjunct of $\mathsf{pushObl}(pushTicket)$. Observe first that if $pushTicket < popTicket$, then xs_i is the empty list since its length is $pushTicket - popTicket = 0$. Thus abstract state of the coarse-grained queue is thus the singleton $[v_s]$. From the collection of pending dequeues in the invariant we get

$$\exists id.\, \mathsf{idsAt}^{\gamma_m}(pushTicket, id) * \left(\models - \precsim_{id} \mathsf{dequeue}_{CG}\ w : -\right).$$

We know that the specification queue contains exactly the value v_s, so we can symbolically execute the right-hand side $\mathsf{dequeue}_{CG}\ w$ and reduce it to the value v_s by using REL-DEQUEUE-DETACHED. This suffices for showing $\mathsf{pushObl}(pushTicket)$. We close the invariant while keeping the persistent points-to predicate for ℓ_{arr} and the turn. We can now simply step through the rest of the code. At $\mathsf{dequeue}_{SEQ}$ we apply the unary specification for it and we have both a turn for it and the resource to show the predicate for the value we enqueue.

REL-SPLIT
$$\frac{\forall id. (\models - \precsim_{id} e_2 : -) \twoheadrightarrow \Delta \models e_1 \precsim_{id} - : \tau}{\Delta \models e_1 \precsim e_2 : \tau}$$

REL-COMBINE
$$\frac{\Delta \models e_1 \precsim e_2 : \tau}{(\models - \precsim_{id} e_2 : -) \twoheadrightarrow \Delta \models e_1 \precsim_{id} - : \tau}$$

REL-RIGHT-LOAD
$$\frac{\ell \hookrightarrow_s v \qquad \models - \precsim_{id} K[\,!\,\ell\,] : -}{\Rrightarrow (\ell \hookrightarrow_s v) * (\models - \precsim_{id} K[\,v\,] : -)}$$

Figure 10. Selected rules for external linearization points.

8.2 Proof of dequeue

For dequeue we prove that $\lambda x.$ dequeue ℓ_{arr} q ℓ_{pop} refines $\lambda x.$ dequeue_{CG} w. We again we symbolically execute the left-hand side and open the invariant around FAA. From the invariant we get the predicate $\ell_{pop} \hookrightarrow popTicket$. Similarly to what happened in enqueue we must close the invariant with $popTicket + 1$. We can now take the turn $\text{turn}_d(\gamma, n)$ for the dequeue of the ($popTicket$ mod q)'th single-element queue with $n = \lfloor popTicket/q \rfloor$. If $popTicket < pushTicket$, then we can take $\text{listAt}^{\gamma_l}(popTicket, v)$ using GHOST-LIST-LOOKUP. This corresponds to the first element in the queue and we can use this to step the specification forward using REL-DEQUEUE-R. Otherwise, the linearization point is external, and we apply the rule REL-SPLIT and introduce $(\models - \precsim_{id} \text{dequeue}_{CG}\ w : -)$. We use IDENTIFIER-DECIDE to get $\text{idsAt}^{\gamma_m}(popTicket, id)$. We transfer both of these into the invariant when we close it. When we close the invariant we can also keep $\text{token}^{\gamma_t}(popTicket)$ and, as it is persistent, the points-to predicate for ℓ_{arr}. Using the latter we can step over the load of the array. Using the former we can apply the specification for dequeue on the single-element queue. From this specification we know that the dequeue returns a value v satisfying $\text{listAt}^{\gamma_l}(popTicket, v)$. We then open the invariant again. Intuitively, at this point an enqueue operation must have carried out dequeue's linearization point. We can conclude that $popTicket < pushTicket$, since we know that the ghost list has a value at index $popTicket$, and that the length of the ghost list is $pushTicket$. Hence the invariant contains $\text{pushObl}(popTicket)$; the first disjunct herein being in contradiction with our $\text{token}^{\gamma_t}(popTicket)$. Note how the token we took when we first opened the invariant now serves as a proof that the corresponding enqueue could not have carried out this obligation by providing the token. Hence we have the content in the second disjunct:

$$\exists id, v, v_s.\ \text{idsAt}^{\gamma_m}(popTicket, id) * (\models - \precsim_{id} v_s : -) * \\ [\![\tau]\!]_\Delta(v, v_s) * \text{listAt}^{\gamma_l}(popTicket, v)$$

We introduce the existentials; by IDENTIFIER-AGREE it is clear that the first must be equal to id. We can now apply REL-COMBINE and step the implementation down to the value v for which we have $[\![\tau]\!]_\Delta(v, v_s)$ and hence these are related.

9 Discussion: Conclusion, Related and Future Work

We now conclude and discuss related and future work along two dimensions: (1) specification and verification of the MPMC queue, and (2) and the extension of ReLoC with support for reasoning about external linearization points. Wrt. (1), to the best of our knowledge, ours is the first formal specification and verification of the MPMC queue. We emphasize that our verification approach is *modular*. For example, our specification for the turn sequencer can also be used to verify other clients than the single-element queue; indeed, in our Coq formalization we have used the turn sequencer to implement and verify a ticket lock. Our implementation of the MPMC queue does abstract over some implementation aspects of the Facebook C++ implementation. In particular, the latter makes use of so-called futex'es; whereas in our implementation we use CAS loops when threads have to wait. Futex'es are a mechanism provided by the operating system, which are more efficient than using CAS loops. However, futex'es are not modelled by any existing concurrent separation logic; future work thus includes extending our operational semantics model to also account for futex'es and to extend Iris and ReLoC with support for reasoning about futex'es.

Wrt. (2), we emphasize that our extensions to ReLoC are generally applicable for reasoning about external linearization points. Indeed in our Coq formalization we have applied our methodology to two other examples: a version of the elimination-backoff stack from [10], and the red flags versus blue flags example from [27]. The most closely related work not already discussed earlier in the paper is Liang and Feng's relational logic [20], which can be used to show refinement for fine-grained concurrent algorithms with non-fixed linearization points, including algorithms with external linearization points. In contrast to Liang and Feng's logic, our extended version of ReLoC supports a more expressive programming language (also including higher-order functions and higher-order state), and perhaps more importantly for the results in this paper, it is fully implemented and verified in Coq. We have taken advantage of the latter by using the Coq infrastructure to mechanize and formally prove the contextual refinement of the MPMC queue. Recently, a variant of Liang and Feng's logic has been formalized in Coq by Zou et. al. [32] and it could be interesting to investigate how a proof of the MPMC queue in that setting would compare with our proof in ReLoC. As indicated by the additional case studies mentioned above, we believe that our extension to ReLoC is sufficiently general to support mechanized verification of a wide range of fine-grained concurrent algorithms with external linearization points, and we hope it will be used for that in future work.

Acknowledgments

We are grateful to Peter O'Hearn from Facebook who suggested to us that the MPMQ queue would be an interesting challenge to specify and verify. We also thank Nathan Bronson and Maged Michael from Facebook for informing us about the history of the MPMC queue development at Facebook. 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] Bodil Biering, Lars Birkedal, and Noah Torp-Smith. 2007. BI-hyperdoctrines, higher-order separation logic, and abstraction. *ACM Trans. Program. Lang. Syst.* 29, 5 (2007), 24. https://doi.org/10.1145/1275497.1275499

[2] Nathan Bronson. 2020. On the origins of the MPMC Queue. Personal Communication.

[3] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. USENIX Association, San Jose, CA, 49–60. https://www.usenix.org/conference/atc13/technical-sessions/presentation/bronson

[4] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. 2013. Views: Compositional Reasoning for Concurrent Programs. In *POPL*. 287–300. https://doi.org/10.1145/2429069.2429104

[5] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew Parkinson, and Viktor Vafeiadis. 2010. Concurrent Abstract Predicates. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings (Lecture Notes in Computer Science)*, Theo D'Hondt (Ed.), Vol. 6183. Springer, 504–528. https://doi.org/10.1007/978-3-642-14107-2_24

[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 Filipović, Peter O'Hearn, Noam Rinetzky, and Hongseok Yang. 2010. Abstraction for concurrent objects. *Theoretical Computer Science* 411, 51-52 (2010), 4379–4398. https://doi.org/10.1016/j.tcs.2010.09.021

[8] Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2018. ReLoC: A Mechanised Relational Logic for Fine-Grained Concurrency. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, Anuj Dawar and Erich Grädel (Eds.). ACM, 442–451. https://doi.org/10.1145/3209108.3209174

[9] 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

[10] Danny Hendler, Nir Shavit, and Lena Yerushalmi. 2004. A Scalable Lock-Free Stack Algorithm. In *SPAA*. 206–215. https://doi.org/10.1145/1007912.1007944

[11] Maurice Herlihy and Jeannette Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *TOPLAS* 12, 3 (1990), 463–492. https://doi.org/10.1145/78969.78972

[12] Bart Jacobs and Frank Piessens. 2011. Expressive modular fine-grained concurrency specification. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 271–282. https://doi.org/10.1145/1926385.1926417

[13] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 256–269. https://doi.org/10.1145/2951913.2951943

[14] 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

[15] 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

[16] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 637–650. https://doi.org/10.1145/2676726.2676980

[17] 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

[18] Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The Essence of Higher-Order Concurrent 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, 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

[19] 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. http://dl.acm.org/citation.cfm?id=3009855

[20] 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

[21] Maged Michael and Michael Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *PODC*. 267–275. https://doi.org/10.1145/248052.248106

[22] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. 2014. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings (Lecture Notes in Computer Science)*, Zhong Shao (Ed.), Vol. 8410. Springer, 290–310. https://doi.org/10.1007/978-3-642-54833-8_16

[23] Matthew Parkinson and Gavin Bierman. 2005. Separation logic and abstraction. 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, 247–258. https://doi.org/10.1145/1040305.1040326

[24] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. Mechanized verification of fine-grained concurrent programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Steve Blackburn (Eds.). ACM, 77–87. https://doi.org/10.1145/2737924.2737964

[25] Kasper Svendsen and Lars Birkedal. 2014. Impredicative Concurrent Abstract Predicates. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings (Lecture Notes in Computer Science)*, Zhong Shao (Ed.), Vol. 8410. Springer, 149–168. https://doi.org/10.1007/978-3-642-54833-8_9

[26] Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 377–390. https://doi.org/10.1145/2500365.2500600

[27] Aaron 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

[28] Aaron Turon and Mitchell Wand. 2011. A separation logic for refining concurrent objects. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 247–258. https://doi.org/10.1145/1926385.1926415

[29] Viktor Vafeiadis. 2008. *Modular fine-grained concurrency verification*. Ph.D. Dissertation. University of Cambridge.

[30] Viktor Vafeiadis and Matthew Parkinson. 2007. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007, Proceedings (Lecture Notes in Computer Science)*, Luís Caires and Vasco Thudichum Vasconcelos (Eds.), Vol. 4703. Springer, 256–271. https://doi.org/10.1007/978-3-540-74407-8_18

[31] Simon Friis Vindum and Lars Birkedal. 2020. Proof Pearl: Contextual Refinement of the Michael-Scott Queue. https://www.cs.au.dk/~birke/papers/2021-ms-queue.pdf Manuscript under submission.

[32] Mo Zou, Haoran Ding, Dong Du, Ming Fu, Ronghui Gu, and Haibo Chen. 2019. Using concurrent relational logic with helpers for verifying the AtomFS file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, Tim Brecht and Carey Williamson (Eds.). ACM, 259–274. https://doi.org/10.1145/3341301.3359644