Mechanized Verification of a Fine-Grained Concurrent Queue from Meta'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 Meta's C++ library Folly of core infrastructure components. The queue is highly optimized, practical, and used by Meta in production where it scales to thousands of consumer and producer threads. We present an implementation of the algorithm in an ML-like language and formally prove that it is a contextual refinement of a simple coarse-grained queue (a property that implies that the MPMC queue is linearizable). We use the ReLoC relational logic and the Iris program logic to carry out the proof and to mechanize it in the Coq proof assistant. The MPMC queue is implemented using three modules, and our proof is similarly modular. By using ReLoC and Iris's support for modular reasoning we verify each module in isolation and compose these together. A key challenge of the MPMC queue is that it has a so-called *external linearization point*, which ReLoC has no support for reasoning about. Thus we extend ReLoC, both on paper and in Coq, with novel support for reasoning about external linearization points.

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

Keywords: separation logic, concurrent data structures, Coq

ACM Reference Format:
Simon Friis Vindum, Dan Frumin, and Lars Birkedal. 2022. Mechanized Verification of a Fine-Grained Concurrent Queue from Meta's Folly Library. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '22), January 17–18, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3497775.3503689

CPP '22, January 17–18, 2022, Philadelphia, PA, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9182-5/22/01.
https://doi.org/10.1145/3497775.3503689

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 program logics for specifying and verifying such algorithms, *e.g.,* [9, 15, 21, 25, 30, 38, 39, 41, 42, 44–46]. In this paper, we present the first formal specification and verification of the highly efficient and practical concurrent multi-producer-multi-consumer queue algorithm found in Meta's open-source library Folly (or, simply, the MPMC queue in the rest of the paper).

The Folly library is an open-source collection of key infrastructure components implemented in C++ and used extensively in production at Meta [32]. The library contains, among many other things, the MPMC queue.[1] The queue was originally developed by Nathan Bronson to connect two thread pools inside TAO, Meta's distributed data store for their social graph [5]. One of the key ideas used in the algorithm is to improve scalability by decreasing the contention found in other lock-free algorithms, such as the Michael-Scott queue [35], by striping the queue across q "smaller" sub-queues. To avoid the overhead of maintaining q sub-queues, the striping is taken to the extreme by letting each sub-queue store only a single element. These *single-element queues* can then be simpler and faster. In fact, they are implemented merely as a reference to a value and a so-called *turn sequencer*. The latter is a synchronization mechanism used by the single-element queue to guard access to its value. The enqueue and dequeue operations on the MPMC queue are delegated to one of the single-element queues by taking a ticket from one of two ticket dispensers using an atomic increment (FAA). After receiving a ticket, up to q separate enqueue or dequeue operations can proceed in parallel, completely independent of each other as they operate on different single-element queues. The FAA instruction thus becomes the main point of contention, but since an FAA instruction (unlike CAS) always succeeds, this design, in the words of Bronson, "makes contention count" as its cost always pays off in significant progress being made in the algorithm [4]. Altogether, this makes the queue scalable to hundreds of thousands of producer and consumer threads.

More concretely, in this paper we present an implementation of the MPMC queue and all its components in an ML-like

[1] The source code is available online at https://github.com/facebook/folly/blob/main/folly/MPMCQueue.h.

language with concurrency primitives. The implementation captures the essence and the key verification challenges of the algorithm while eliding some of the low-level details of the original C++ implementation. We prove that the MPMC queue *contextually refines* a coarse-grained concurrent queue. The coarse-grained queue uses a lock to ensure that only one thread at a time access the queue. We take this simple queue to be the *specification* of a queue and the MPMC queue to be an *implementation* of the specification. Informally, the contextual refinement property then means that in any program we may replace uses of the "obviously correct" coarse-grained concurrent queue with the more efficient, but also more complicated, MPMC queue, without changing the observable behavior of the program. More precisely, an expression e contextually refines another expression e', if for all contexts C of a ground type, if $C[e]$ terminates with a value, then there exists an execution of $C[e']$ that terminates with the same value.

We prove the contextual refinement using the recently proposed relational logic ReLoC [15, 16], which builds on top of the Iris separation logic [22, 23, 25, 28] and greatly simplifies proofs of contextual refinement by offering rules that allows one to reason about refinements at a high level of abstraction. Additionally, it is mechanized in Coq and allowed us to develop our mechanized proof interactively using the Iris proof mode [27, 29].

To verify a fine-grained concurrent algorithm, one of the key steps is to identify the *linearization points* of its operations: the point during execution where the operation "appears to take place". In our analysis of the MPMC queue we discover that, in some cases, the linearization point of the dequeue operation is *external.* A linearization point is external if it happens during the execution of another operation. For dequeue, its linearization point may happen within the execution of an enqueue operation, which is not immediately obvious by looking at the code. As we explain in detail later, the external linearization points arise because the algorithm, in contrast to other fine-grained concurrent queues, is not entirely non-blocking: if all the single-element queues are full (resp. empty) then enqueue (resp. dequeue) is blocked.

One may categorize linearization points into three classes [11]: fixed, future-dependent, and external. The first version of ReLoC [15] had support for reasoning about fixed linearization points only, and ReLoC Reloaded [16] added support for future-dependent linearization points, through its use of Iris-style prophecy variables [24]. However, we observe that neither version of ReLoC supports reasoning about external linearization points. The high-level reason is that ReLoC ties the state of the implementation with the state of the specification as a single judgment. At an external linearization point (in our case in dequeue) the state of the specification must be transferred to the other operation where the linearization point takes place (in our case enqueue). This is not possible with ReloC's existing rules. Hence, to verify the MPMC queue we extend ReLoC with new proof rules and generalize its existing proof rules to be able to reason about external linearization points. The extension is simple but elegant and "completes the picture" by making ReLoC able to handle all three classes of linearization points. External linearization points often occur due to *helping* and our extension makes ReLoC able to handle these concurrent data structures with helping as well.

As mentioned, the MPMC queue is implemented as three submodules: the MPMC queue is implemented using the single-element queue, which is implemented using the turn-sequencer. A strength of our approach is that our contextual refinement proof is similarly modular: it makes use of (unary) Hoare-style specifications of the turn-sequencer and the single-element queue. Here we leverage the fact that ReLoC allows for compositional reasoning and that it, following [41], 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 [15, Section 7.4]. We thus end up not only with a refinement proof of the MPMC queue but also with reusable specifications for the single-element queue and the turn sequencer.

To arrive at sufficiently composable specifications we make use of a proof pattern involving a resource algebra over *infinite* sets, to keep track of which turns are "still available" (see Section 4). The idea of using infinite structures to improve composability is well-known in the context of functional programming [20]. In our case, the specification of the turn-sequencer supplies its client with an *infinite* set of turns and the specification of the single-element queue gives its client two infinite sets of tickets. This approach greatly simplifies the proofs and makes it possible to reason about the single-element queue in the refinement proof with the details of the turn sequencer having been abstracted away. We believe this proof pattern could also be used to simplify reasoning about other algorithms based on these components, and have used it to additionally verify a ticket lock based on the turn sequencer.

Another challenge in verifying the MPMC queue is that its physical state (*i.e.,* the actual content in the underlying array) does not immediately determine the *abstract state* of the queue (*i.e.,* the state that is observable through the queue interface). In particular, a value may be present in the physical state of the queue without actually being in the queue (*i.e.,* not observable with a dequeue operation), and vice versa. This lies in contrast with other data structures, even those with non-fixed linearization points (such as the Herlihy-Wing queue [19] and the Michael-Scott queue [35]).

In summary, we believe that verifying the MPMC queue serves as an interesting case study, as it is challenging to verify, used at scale in the industry, has not been treated in the literature before, and it provides motivation for extending ReLoC with support for external linearization points.

Outline and contributions.

- Since the MPMC queue has not been treated in the literature before, we give a detailed description of it (Section 2).
- We informally analyze the linearization points of the MPMC queue and observe that one of them is external (Section 3).
- We define and prove Hoare-style specifications for the turn sequencer and single-element queue (Section 4).
- We show that the MPMC queue contextually refines a coarse-grained queue. (Sections 5 and 6). Our proof is *modular* and makes use of the aforementioned Hoare-style specifications for the submodules.
- We explain why prior versions of ReLoC can not handle external linearization points and extend ReLoC, both on paper and in Coq, with support (including tactics) for reasoning about external linearization points (Section 7).
- We have formalized all the results in this paper, and two additional examples of algorithms with external linearization points in the Coq proof assistant [48]. The formalization is part of the ReLoC git repository and can be found online at https://gitlab.mpi-sws.org/iris/reloc/-/tree/master/theories/examples/folly_queue. The version that we specifically refer to in this paper corresponds to the commit with the git hash `b6df47f9`.

We discuss related and future work in Section 8.

2 The Folly MPMC queue

We now 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 thread waits for the *n*th turn, then it will only proceed once all the preceding turns have been completed. 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 (SEQ) is a queue with a capacity of one. 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).

Similarly to the turn sequencer, the SEQ's operations take a turn as an argument, however the turns are separate for enqueue and dequeue. The turn argument specifies the order of the operations: an enqueue or dequeue operation is carried out only after all operations with a lower number have been carried out. For an enqueue and a dequeue operation with the same turns, the enqueue is carried out first. This ordering ensures that when an enqueue operation is carried out, the queue is always empty, and when dequeue is run the queue is non-empty.

The SEQ is implemented as a reference to an option type, protected by a single turn sequencer. 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 $\mathsf{enqueue}_{SEQ}$ ($\mathsf{dequeue}_{SEQ}$, 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.,* $3/2 = 1$). The Λ is a type abstraction (or a generic) making the queue polymorphic in the type of values it can store.

Upon initialization, an array of length q is created, with each entry containing a SEQ. 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 keep track of tickets for the enqueue operation, and the second does the same for the dequeue operation.

```
newTS () = ref(0)
complete ts turn = ts ← turn + 1; ()
wait ts turn =
  let turn′ = ! ts in
  if (turn′ = turn) then ()
  else wait ts turn
queue_SEQ () = (newTS (), ref(None))
enqueue_SEQ (ts, r) enqTurn v =
  let turn = enqTurn ∗ 2 in
  wait ts turn;
  r ← Some(v);
  complete ts turn
dequeue_SEQ (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.

```
queue_MPMC q = Λ.
  let slots = arrayInit q queue_SEQ 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
  enqueue_SEQ (slots[idx]) ticket v
dequeue slots q popTicket =
  let t = FAA(popTicket, 1) in
  let idx = t mod q in
  let ticket = t/q in
  dequeue_SEQ (slots[idx]) ticket v
```

(b) MPMC queue.

```
queue_CG = Λ.
  let w = (newlock (), ref([])) in
  (λv. enqueue_CG w v,
   λx. dequeue_CG w)
enqueue_CG (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
dequeue_CG (lk, hd) =
  acquire lk;
  match ! hd with
  | [] ⇒ assert(false)
  | h :: t ⇒ hd ← t;
             release lk;
             h
```

(c) Coarse-grained queue.

Figure 1. Implementation of the various data structures.

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. From this ticket, we calculate an index ($t \bmod q$) in the array for a SEQ. Then, enqueue writes an element into the SEQ 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 SEQ and returns this value.

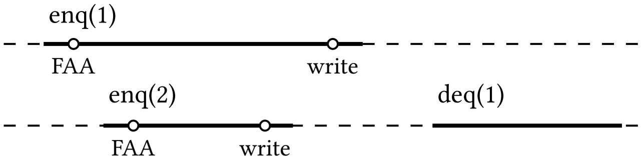
2.4 Relationship to original C++ code

Our implementation of the MPMC queue in ReLoC's ML-like language is faithful to the original algorithm, but does omit some low-level details of the original C++ implementation.

- The C++ implementation takes into account the C++ relaxed-memory model whereas the memory model of ReLoC's ML-like language is sequentially consistent. ReLoC does not support weak memory so verifying the MPMC queue in a weak memory setting would have required a different verification methodology.
- The C++ turn sequencer gracefully handles integer overflow of the turn counter. As the ML-like language included with ReLoC only support unbounded integers our implementation does not handle overflow.
- When waiting for a turn, the C++ turn sequencer uses a heuristic consisting of spinning with a back-off and suspending the thread (using futexes [14]) for increased performance. Our implementation only uses spinning. This difference only affects efficiency and not the safety or linearizability of the algorithm. Additionally, to manage the use of futexes the integer in the turn sequencer stores not only the current turn but also uses some bits to manage sleeping threads. Due to this, the turn is incremented using compare-and-set and not FAA as in our implementation.
- The C++ implementation supports additional operations in addition to the queue operations dequeue and enqueue. For instance, an enqueue operation that fails instead of blocking when the queue is full.
- The use of closures in our implementation can be seen as corresponding to the use of objects in C++.

3 Linearizability of the MPMC queue

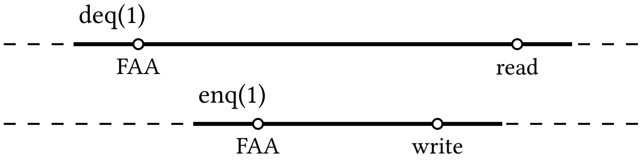
In this section, we analyze the MPMC queue informally and identify its linearization points. As a first guess, one might think that the linearization point for enqueue is when enqueue writes its value into the SEQ and, similarly, for dequeue when it reads the value from the SEQ. After all, these are 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 receives ticket 0. Afterward the second enqueue executes its FAA, receives 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 operation happens at the FAA. If an enqueue operation receives a ticket i, then clearly the value that it inserts 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 sense 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 operation 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 the 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, the enqueued element becomes a part of the abstract state, before it is even written into the array. Symmetrically, when a dequeue receives a ticket, it removes an element 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 is not related directly to the physical state of the array. The abstract state is, however, directly related to the values 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 define suitable Hoare triple specifications for the turn sequencer and the SEQ. We also sketch how these are proved. We emphasize that the proof of the SEQ 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 SEQ. Thus our specifications and proofs are *modular*, and we observe that to prove contextual refinement for the MPMC queue, a unary specification for the SEQ suffices.

4.1 Turn Sequencer

As mentioned earlier, the turn sequencer is a mechanism for mutual exclusion. Therefore, our specification of the turn sequencer (shown in Figure 2) is an extension of a typical concurrent separation logic specification for a lock [2, 17], and the verification process is similar to the verification of a ticket-based lock [31, Section 2.2]. There are two key differences though. The first difference is that it is up to the client of the turn sequencer to ensure that the turns are

Turn Sequencer

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

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

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

Single-Element Queue

$$\{\mathsf{True}\}\ \mathsf{queue}_{SEQ}\ ()\ \left\{v.\exists \gamma.\ \begin{array}{c} \mathrm{isSEQ}(\gamma, Q, v) * \\ \mathrm{turns}_e(\gamma, 0) * \mathrm{turns}_d(\gamma, 0) \end{array}\right\}$$

$$\begin{array}{l} \{\mathrm{isSEQ}(\gamma, Q, v) * \mathrm{turn}_e(\gamma, n) * Q(n, x)\} \\ \quad \mathsf{enqueue}_{SEQ}\ v\ n\ x \\ \{\mathsf{True}\} \end{array}$$

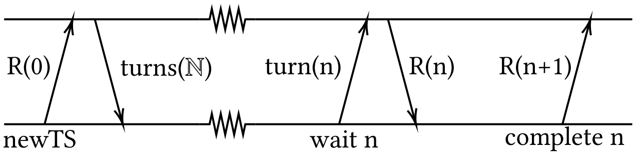
$$\{\mathrm{isSEQ}(\gamma, Q, v) * \mathrm{turn}_d(\gamma, n)\}\ \mathsf{dequeue}_{SEQ}\ v\ n\ \{x.Q(n, x)\}$$

Figure 2. Unary specifications for turn sequencer and SEQ.

used correctly. For instance, wait should never be invoked with a past turn. The second difference is that the resource protected by the turn sequencer is indexed by a turn number, which allows for a more dynamic treatment of resources protected behind a critical section. In some sense, this makes the specification for the turn sequencer stronger than that for a lock, and in our Coq formalization we have implemented and verified a lock based on the turn sequencer.

The specification uses two predicates "close" and "isTS", which are abstract to clients of the specification (as in [1, 37]). The latter, "isTS", is the representation predicate. It is *persistent*, which intuitively means that, unlike other separation logic propositions, it is freely duplicable and not consumed by preconditions.

The predicate R describes the resource that the turn sequencer protects. Whereas a lock protects a resource $R : iProp$, the turn sequencer protects a $\mathbb{N}$-indexed *family* of resources, that is, $R : \mathbb{N} \to iProp$, where the index represents the current turn. This generalization of the protected resource is possible since the turn sequencer guarantees to run clients in the order of their turns. When it becomes a client's 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 the next thread in line (which is waiting for the turn $n+1$).

TURN-ALLOC

$$\frac{X \subseteq \mathbb{N}}{|\!\!\Rrightarrow \exists \gamma.\ \mathrm{turns}(\gamma, X)}$$

TURN-DISJ

$$\frac{\mathrm{turns}(\gamma, X) \qquad \mathrm{turns}(\gamma, Y)}{X \cap Y = \emptyset}$$

TURN-SEP

$$\frac{X \cap Y = \emptyset}{\mathrm{turns}(\gamma, X) * \mathrm{turns}(\gamma, Y) \dashv\vdash \mathrm{turns}(\gamma, X \cup Y)}$$

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 [8, 25, 36]. 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. For that purpose, we use a predicate $\mathrm{turns}(\gamma, X)$ that denotes ownership over the set of turns $X \subseteq \mathbb{N}$, and the singular $\mathrm{turn}(\gamma, n) \triangleq \mathrm{turns}(\gamma, \{n\})$ that 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 sets of turns implies that the sets are disjoint (TURN-DISJ). Ownership over two disjoint sets of turns is equivalent to ownership of their union (TURN-SEP).

As depicted in the diagram above, when a client creates a new turn sequencer, it acquires ownership over *all* turns: $\mathrm{turns}(\gamma, \mathbb{N})$. To call wait for a turn n the client must own $\mathrm{turn}(\gamma, n)$, the ownership of which is then transferred into the turn sequencer, ensuring that the client can only wait for the same turn once. This is necessary for safety of the turn sequencer, as previously mentioned.

Finally, when a client acquires the current turn, it gets $\mathrm{close}(v, n)$, 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:

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

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

The predicate "isTS" is defined as an *invariant.* An invariant $\boxed{P}^{\mathcal{N}}$ represents the knowledge that the proposition P always holds. Since an invariant is knowledge and not a resource that one owns, this definition satisfies the previously mentioned property that "isTS" is persistent.

With these definitions, we now sketch how the specifications are proved.

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

Similar to the specification for the turn sequencer, in the specification for the SEQ (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 SEQ 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 $Q : \mathbb{N} \rightarrow \mathit{Val} \rightarrow \mathit{iProp}$. 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). First, we 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:

$$\begin{aligned}
\mathrm{turns}_e(\gamma, n) &\triangleq \mathrm{turns}(\gamma, \{m \in \mathbb{N} \mid \mathrm{even}(m) \wedge 2n \quad \leq m\}) \\
\mathrm{turns}_d(\gamma, n) &\triangleq \mathrm{turns}(\gamma, \{m \in \mathbb{N} \mid \mathrm{odd}(m) \wedge 2n + 1 \leq m\}) \\
\mathrm{turn}_e(\gamma, n) &\triangleq \mathrm{turn}(\gamma, 2n) \\
\mathrm{turn}_d(\gamma, n) &\triangleq \mathrm{turn}(\gamma, 2n + 1)
\end{aligned}$$

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

Next, we define the representation predicate isSEQ by instantiating the turn sequencer specification:

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

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

The predicate $\mathrm{isSEQ}(\gamma, Q, v)$ states that the value v making up the SEQ is a pair of a location ℓ and a turn sequencer ts. 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 SEQ. This is why R_{SEQ} applies Q to $(n-1)/2$.

With these definitions, the *SEQ* specification can be derived from the turn sequencer specification.

4.3 Ghost state for Turns and Tickets.

We now detail the construction of the ghost state used to represent turns. This section can be skipped—understanding the derived rules presented in the previous two sections suffices for the rest of the paper.

In Iris ghost state is represented using a form of partial commutative monoids called a *resource algebra.* The monoid operation $(\cdot)$ combines elements of the resource algebra and a subset of elements $\mathcal{V}$ are *valid.* In the logic the ownership assertion $\boxed{a}^{\gamma}$ denotes ownership over an element a of some resource algebra for a ghost name γ. Any valid element can be allocated for a fresh ghost name γ (GHOST-ALLOC), ownership of two elements combine into ownership of their combination per the operation (OWN-OP), and owned elements are always valid (OWN-OP).

We want to represent ownership of, potentially, infinite sets of turns. Since ownership of a turn should be exclusive, we want the combination of two sets to be invalid if the sets are not disjoint. The naive approach of letting the elements

$$\textsc{Ghost-alloc}\quad \frac{a \in \mathcal{V}}{\Rrightarrow \exists \gamma.\, \boxed{a}^{\gamma}} \qquad \textsc{Own-op}\quad \frac{}{\boxed{a}^{\gamma} * \boxed{b}^{\gamma} \dashv\vdash \boxed{a \cdot b}^{\gamma}} \qquad \textsc{Own-valid}\quad \frac{}{\boxed{a}^{\gamma} \vdash a \in \mathcal{V}}$$

$$\textsc{set-alloc}\quad \frac{X \subseteq \mathbb{N}}{\Rrightarrow \exists \gamma.\, \boxed{\mathbf{1}_X}^{\gamma}} \qquad \textsc{set-sep}\quad \frac{X \cap Y = \emptyset}{\boxed{\mathbf{1}_X}^{\gamma} * \boxed{\mathbf{1}_Y}^{\gamma} \dashv\vdash \boxed{\mathbf{1}_X \cup \mathbf{1}_Y}^{\gamma}}$$

$$\textsc{set-disj}\quad \frac{X \cap Y \neq \emptyset \qquad \boxed{\mathbf{1}_X}^{\gamma} \qquad \boxed{\mathbf{1}_Y}^{\gamma}}{\mathsf{False}}$$

Figure 4. Rules for ghost state and the resource algebra of (infinite) sets.

of the resource algebra be sets and defining the operation as

$$A \cdot B \triangleq \begin{cases} A \cup B & \text{if } A \cap B = \emptyset \\ \bot & \text{otherwise} \end{cases}$$

where $\bot$ is invalid, does not work The operation must be computable, but determining if two arbitrary infinite sets intersect is not.[2]

Instead, we represent sets using a function resembling a characteristic function. We assemble a resource algebra using three standard resource algebras: the exclusive resource algebra, the option resource algebra, and the resource algebra of functions.

$$\mathit{InfSet}(X) = X \to \mathit{Option}(\mathit{Ex}(1))$$

For the resource algebra of functions, the operation is defined point-wise as $(f \cdot g)(a) = f(a) \cdot g(a)$. Its elements are valid $f \in \mathcal{V}$ if and only if $f(a) \in \mathcal{V}$ for all a in the function's domain. The codomain $\mathit{Option}(\mathit{Ex}(1))$ has two valid elements none and some(ex()); and one invalid element some($\bot$). These combine in the following way:

$$\begin{aligned} \mathsf{none} \cdot \mathsf{some}(\mathsf{ex}()) &= \mathsf{some}(\mathsf{ex}()) \\ \mathsf{some}(\mathsf{ex}()) \cdot \mathsf{some}(\mathsf{ex}()) &= \mathsf{some}(\mathsf{ex}() \cdot \mathsf{ex}()) = \mathsf{some}(\bot) \end{aligned}$$

For any $A \subseteq X$ we can define an element $\mathbf{1}_A \in \mathit{InfSet}(X)$ as

$$\mathbf{1}_A(a) = \begin{cases} \mathsf{some}(\mathsf{ex}()) & \text{if } a \in A \\ \mathsf{none} & \text{if } a \notin A \end{cases}$$

The idea being that $\mathbf{1}_A$ serves as a sort of characteristic function. Given two *disjoint* sets A and B it is then the case that $\mathbf{1}_A \cdot \mathbf{1}_B = \mathbf{1}_{A \cup B}$. On the other hand, given A and B that are not disjoint, then for $a \in A \cap B$ it is the case that $(\mathbf{1}_A \cdot \mathbf{1}_B)(a) = \bot$ and hence the combination is invalid. The three last rules in Figure 4 then follow immediately.

With this in place "turns" is defined simply as

$$\mathsf{turns}(\gamma, X) \triangleq \boxed{\mathbf{1}_X}^{\gamma}.$$

With this definition the previously seen rules for turns in Figure 3 follow from the rules in Figure 4.

5 Proof of Contextual Refinement

As mentioned, our main result is that the MPMC queue is a contextual refinement of a coarse-grained queue. We can succinctly state this as the following ReLoC proposition:

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

for any $q > 0$. In such a *refinement judgment* the left expression is called the *implementation* and the right expression the *specification*.

A refinement judgment is manipulated using ReLoC's high-level rules. While the details are not important, a few such rules appear in Figure 5. The key principle is that the implementation and specification can be *symbolically executed*, similarly to how it is done in a unary program logic with a Hoare triple or a weakest precondition judgment. The rules rel-load-l and rel-load-r show how to symbolically execute a load operation in the implementation and specification respectively. When the implementation and specification are both values one must show that the values are related. What this means depends on the type of the values; for integers, for instance, it means that they are equal.

Proofs of refinements, like the one above, consist of three parts: (a) Symbolically execute the initialization (*i.e.,* the constructor) of the implementation and specification, and collect the resources. (b) Establish an invariant, using the resources obtained from the first step. The invariant typically relates the internal states of the data structures on the both sides of the refinement. Picking the right invariant is the key to the proof, and we discuss it in details in Section 6. (c) Using the invariant, verify refinement of each operation that is part of the data structure. In this stage we verify separately that MPMC's dequeue operation refines the coarse-grained queue's dequeue operation, and similarly for the enqueue operation.

For the first step, due to the polymorphic type $\forall \alpha.(1 \to \alpha) \times (\alpha \to 1)$ of the queue, we must assume a binary predicate τ_i that represents what it means for values of a type τ to be related. Then, we symbolically execute the initialization code for the MPMC queue and obtain the resources:

$$(\ell_{push} \hookrightarrow 0) * (\ell_{pop} \hookrightarrow 0) * (\ell_{arr} \hookrightarrow_{*} \mathsf{map}\ \pi_2\ \mathit{SEQs}) *$$

$$\mathop{\Huge *}_{(v,\gamma) \in \mathit{SEQs}} \mathrm{isSEQ}(\gamma, Q, v) * \mathsf{turns}_e(\gamma, 0) * \mathsf{turns}_d(\gamma, 0).$$

[2] This type of ghost state was used in the GPS logic [43] to verify a ticket lock. But, since the proof was on paper only, the non-computability of the operation was not an issue. A variant of the GPS proof was later mechanized in the iGPS logic [26], using a type of ghost state based on cofinite sets and not arbitrary infinite sets.

$$\textsc{rel-lam}\quad \frac{\square\left(\forall v_1, v_2.\ [\![\tau]\!]_\Delta(v_1, v_2) \mathrel{-\!\!*} \Delta \models (\lambda x_1.\ e_1)\ v_1 \precsim (\lambda x_2.\ e_2)\ v_2 : \sigma\right)}{\Delta \models (\lambda x_1.\ e_1) \precsim (\lambda x_2.\ e_2) : \tau \to \sigma}$$

$$\textsc{rel-load-r}\quad \frac{\ell \hookrightarrow_{\mathrm{s}} v \qquad \ell \hookrightarrow_{\mathrm{s}} v \mathrel{-\!\!*} \models_{\mathcal{E}} e_1 \precsim K[\,v\,] : \tau}{\models_{\mathcal{E}} e_1 \precsim K[\,!\,\ell\,] : \tau}$$

$$\textsc{rel-load-l}\quad \frac{\ell \hookrightarrow v \qquad \ell \hookrightarrow v \mathrel{-\!\!*} \Delta \models K[\,v\,] \precsim e_2 : \tau}{\Delta \models K[\,!\,\ell\,] \precsim e_2 : \tau}$$

$$\textsc{rel-queue-r}\quad \frac{\forall w.\ \mathrm{I}_{\mathrm{CG}}(w, \vec{x}) \mathrel{-\!\!*} \models t \precsim K[\,w\,] : \tau}{\models t \precsim K[\,(\mathsf{newlock}\ (), \mathsf{ref}([\,]))\,] : \tau}$$

$$\textsc{rel-dequeue-r}\quad \frac{\mathrm{I}_{\mathrm{CG}}(w, v :: \vec{x}) \qquad (\mathrm{I}_{\mathrm{CG}}(w, \vec{x}) \mathrel{-\!\!*} \models t \precsim K[\,v\,] : \tau)}{\models t \precsim K[\,\mathsf{dequeue}_{\mathrm{CG}}\ w\,] : \tau}$$

Figure 5. ReLoC rules (selection).

The first two points-to predicates are from allocating *pushTicket* and *popTicket*, while the remaining are from allocating the array and the SEQs it contains. We obtain a pointer ℓ_{arr} to the array. For each element of the array, we invoke queue_{SEQ} and, using its specification from Figure 2, obtain the value v in the array that satisfies $\mathrm{isSEQ}(\gamma, Q, v) * \mathrm{turns}_e(\gamma, 0) * \mathrm{turns}_d(\gamma, 0)$ for some ghost name γ and a predicate Q of our choosing. The list *SEQs* contain the value and ghost name for each SEQ. We describe the appropriate choice of the predicate Q in the next section.

For the coarse-grained queue, we symbolically execute its initialization using rel-queue-r and obtain the resource $\mathrm{I}_{\mathrm{CG}}(w, [])$. The abstract predicate $\mathrm{I}_{\mathrm{CG}}(w, xs)$ states that w is a coarse-grained queue containing the elements xs.

With these resources we have to prove the remainder of the refinement:

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

Naturally, it suffices to show that each operation refines its coarse-grained counterpart. To this end, we use the rule rel-lam, which intuitively states that two functions are related if they *always* (indicated by the □) evaluate to related values when given related input. This reflects that for two implementations to be related they have to be indistinguishable in any context – including a context that calls the functions several times, potentially in parallel. However, the resources that we obtained from the initialization process cannot be used "as is" as they are ephemeral resources that do not always hold. Hence, we delegate those resources to an *invariant.* The refinement proof of the operations can then proceed by symbolically executing the implementation. Every step can assume and must preserve the invariant. The specification side is stepped forward only at linearization points as it is at these points that the implementation changes its abstract state and hence what specification side queue it corresponds to. At the linearization points we thus apply rules such as rel-dequeue-r. We do not explain the refinement proof of the operations in any more detail as defining a suitable invariant is the most challenging part of the refinement proof and is explained in the next section. However, in Section 7 we explain how the external linearization point is handled using our extension to ReLoC.

6 Invariant for Refinement Proof

The invariant we use is shown in Figure 6. It is non-trivial and key to the refinement proof so we devote this section to explain its parts. Overall, the invariant keeps track of the physical state of the queues, ensures that the MPMC queue represents a logic-level list of values corresponding to the coarse-grained queue, manages the turns for all the SEQs, 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).

We now cover each different annotated part of Figure 6 in turn.

Relation to the coarse-grained queue. The existentially quantified lists of values xs_{i} and xs_{s} represent the abstract state of the MPMC queue and the coarse-grained queue respectively. The state of the coarse-grained queue is tied to xs_{s} by $\mathrm{I}_{\mathrm{CG}}(w, xs_{\mathrm{s}})$ and xs_{i} is tied to the MPMC queue by the rest of the invariant. The separating conjunction over the two lists thus ensures that the abstract states of the two queues are always related at type τ_i. For example, if we store integers in the queue, then the separating conjunction states that the xs_{s} and xs_{i} both contain the same integers.

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 $\hookrightarrow_*^{\square}$ [47].

Ghost list. We previously explained how the physical state of the queue reveals very little about the actual values stored in the queue. To connect the physical and abstract states, we use a *ghost list m.* It contains *all* values that have been enqueued, in particular, this includes both values that are no longer and not yet physically 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

$I(\tau_i, \gamma_t, \gamma_m, \gamma_l, q, \ell_{pop}, \ell_{push}, \ell_{arr}, SEQs, w) \triangleq \exists xs_{\mathrm{i}}, xs_{\mathrm{s}} \in List(Val), popTicket, pushTicket \in \mathbb{N}, m \in List(Val).$

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

$$\underbrace{\begin{array}{c}\mathrm{tokensFrom}^{\gamma_t}(\max(popTicket, pushTicket)) * \mathrm{ids}^{\gamma_m}(popTicket) * \\ \left(\mathop{\Large\ast}_{i=0}^{pushTicket-1} \mathrm{enqueueObl}(i)\right) * \left(\mathop{\Large\ast}_{i=pushTicket}^{popTicket-1} \exists id. \begin{array}{l}\mathsf{idsAt}^{\gamma_m}(i, id) * \\ \models - \precsim_{id} \mathrm{dequeue}_{\mathrm{CG}}\ w : -\end{array}\right)\end{array}}_{\textbf{Handling of external linearization points}} * \underbrace{\mathrm{I}_{\mathrm{CG}}(w, xs_{\mathrm{s}}) * \mathop{\Large\ast}_{(x_i,x_s)\in(xs_{\mathrm{i}},xs_{\mathrm{s}})} \tau_i(x_i, x_s)}_{\textbf{Relation to the coarse-grained queue}}$$

$$\begin{aligned}
\textbf{where}\qquad \mathrm{enqueueObl}(i) &\triangleq \mathsf{token}^{\gamma_t}(i) \lor (\exists id, v_{\mathrm{i}}, v_{\mathrm{s}}.\ \mathsf{idsAt}^{\gamma_m}(i, id) * \mathsf{listAt}^{\gamma_l}(i, v_{\mathrm{i}}) * \tau_i(v_{\mathrm{i}}, v_{\mathrm{s}}) * (\models - \precsim_{id} v_{\mathrm{s}} : -)) \\
I_{SEQ}(i, (\gamma, v)) &\triangleq \mathrm{isSEQ}(\gamma, Q(i), v) * \mathrm{turnCtx}(\gamma, i) \\
\mathrm{turnCtx}(\gamma, i) &\triangleq \mathrm{turns}_e(\gamma, \mathrm{affectingOps}(pushTicket, q)) * \mathrm{turns}_d(\gamma, \mathrm{affectingOps}(popTicket, q)) \\
\mathrm{affectingOps}(ops, q) &\triangleq \lfloor ops/q \rfloor + (\text{if } (i < ops \bmod q) \text{ then } 1 \text{ else } 0) \\
Q(i)(j, v) &\triangleq \mathsf{listAt}^{\gamma_l}(jq + i, v)
\end{aligned}$$

Figure 6. Invariant for the MPMC queue

$$\textsc{ghost-list-alloc}\quad \dfrac{}{\Rrightarrow \exists \gamma_l.\ \mathsf{list}^{\gamma_l}([\,])} \qquad \textsc{ghost-list-append}\quad \dfrac{\mathsf{list}^{\gamma_l}(xs)}{\Rrightarrow \mathsf{list}^{\gamma_l}(xs +\!\!+ [x]) * \mathsf{listAt}^{\gamma_l}(|xs|, x)}$$

$$\textsc{ghost-list-agree}\quad \dfrac{\mathsf{listAt}^{\gamma_l}(i, x) \qquad \mathsf{listAt}^{\gamma_l}(i, x')}{x = x'} \qquad \textsc{ghost-list-lookup}\quad \dfrac{\mathsf{list}^{\gamma_l}(xs) \qquad xs_i = x}{\Rrightarrow \mathsf{list}^{\gamma_l}(xs) * \mathsf{listAt}^{\gamma_l}(i, x)}$$

(a) Rules for the ghost list.

$$\textsc{tokens-alloc}\quad \dfrac{}{\Rrightarrow \exists \gamma_t.\ \mathsf{tokensFrom}^{\gamma_t}(0)} \qquad \textsc{token-exclusive}\quad \dfrac{\mathsf{token}^{\gamma_t}(n) \qquad \mathsf{token}^{\gamma_t}(n)}{\mathsf{False}}$$

$$\textsc{tokens-take}\quad \dfrac{\mathsf{tokensFrom}^{\gamma_t}(i)}{\mathsf{tokensFrom}^{\gamma_t}(i+1) * \mathsf{token}^{\gamma_t}(i)}$$

(b) Rules for tokens.

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

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

(c) Rules for identifier registry.

Figure 7. Ghost state rules.

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 xs is denoted by a proposition $\mathsf{list}^{\gamma_l}(xs)$. Ghost list can grow over time, when the new values are enqueued at the end. This is in fact the *only* way in which the ghost list can change, and that means that once a value is part of the ghost list it says there. To that extent, we have a *persistent* predicate $\mathsf{listAt}^{\gamma_l}(i, x)$, which denotes the knowledge that the ith element of the list (corresponding to the the ith value added to the queue) is x. The ghost list satisfies a number of proof rules presented in Figure 7a; these rules are sufficient to carry out the proof.

In the invariant we can see the ownership of the ghost list ($\mathsf{list}^{\gamma_l}(m)$) of the size *pushTicket* ($|m| = pushTicket$). Moreover, if we remove the first *popTicket* elements from m, then the remaining list is exactly the abstract state of the queue ($\mathrm{drop}(popTicket, m) = xs_{\mathrm{i}}$). 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 the Single-Element Queues. For each of the q single-element queues in the array the invariant needs to include the invariant for the SEQ and to manage its turns.

We need to instantiate the invariant for each SEQ with the predicate Q that holds for the values in it. Recall that Q is parameterized both by the value in the queue and its corresponding turn. We use this to define a Q that relates the value in the queue to the "right" value in the ghost list:

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

were q is the capacity of the queue and $0 \leq i < q$ is the index of the particular SEQ. For the jth element v added to this SEQ we can then calculate the position of this element in the whole queue as $jq + i$, which we record using the ghost list.

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

Handling of external linearization points. 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 *tokens*—custom ghost state theory similar to the one that we constructed for turns earlier. The rules for this ghost state are in Figure 7b. The ith dequeue or enqueue that comes first will be able to take the token $\mathsf{token}^{\gamma_t}(i)$. Hence, owning $\mathsf{token}^{\gamma_t}(i)$ proves that an operation came before its corresponding counterpart. The invariant owns all the tokens where neither operation has taken a ticket:

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

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 $pushTicket + 1$ 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 ith token.

Identifier registry. Concretely, for enqueue to carry out its 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 which is explained in greater detail in Section 7 and the approach here is an instance of the general proof pattern identified in Section 7.1. 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 7c. 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

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

Since enqueue closes the invariant with $pushTicket + 1$, it must show $\mathrm{enqueueObl}(\gamma_l, \gamma_t, \gamma_m, pushTicket)$. Hence, one should think of $\mathrm{enqueueObl}(\gamma_l, \gamma_t, \gamma_m, i)$ as something which enqueue is *obliged to produce* when it takes the ith ticket. Since the proposition enqueueObl 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 $\mathrm{token}^{\gamma_t}(pushTicket)$, 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, which involves carrying out the dequeue's linearization point.

7 Extending ReLoC with Support for External Linearization Points

As mentioned earlier, to show that an operation refines its specification with ReLoC, one symbolically executes the implementation up to its linearization point. At the linearization point, the specification is then symbolically executed to reflect the change in the state of the implementation at the linearization point. However, for an external linearization point this approach does not work as the linearization point does not happen during the symbolic execution of the operation; instead, it happens during the symbolic execution of some other operation. Intuitively, it is when we symbolically execute this second operation that we should symbolically execute the specification. This kind of reasoning is not supported by the current ReLoC rules.

To support such reasoning we extend ReLoC with additional rules, a selection of which is shown in Figure 8. We explain how these rules are used by using the external linearization point in the MPMC queue as an example.

When we show the dequeue refinement, we symbolically execute the implementation until we reach the expression $\mathsf{FAA}(popTicket, 1)$. At this point, if $pushTicket \leq popTicket$ then the linearization point is external, and the specification should be symbolically executed during the corresponding enqueue operation. To this end, we apply the rule rel-split which *splits* a refinement judgment into a *left refinement* of the form $\models e_1 \precsim_{id} - : \tau$ and a *right refinement* of the form $\models - \precsim_{id} e_2 : -$. These represent the state of the implementation and the specification, respectively. When we split a refinement judgment, we naturally want to keep track of the fact that the two parts originate from the same refinement judgment. The split refinement judgments is therefore parameterized by an identifier id from an opaque set Id of identifiers. Since the right refinement ($\models - \precsim_{id} e_2 : -$) appears on the left-hand side of a wand $-\!\!*$ in rel-split we can assume it as a *resource*.[3] Hence, after applying rel-split we obtain the right refinement $\models - \precsim_{id} \mathrm{dequeue}_{\mathrm{CG}}\ w : -$ for some id as a proposition. We transfer this right refinement into the invariant, as described in the previous section.

Our goal is now a left refinement with the state of the implementation. To be able to symbolically execute the left refinement, we have generalized all the rules in ReLoC for symbolically executing the implementation in a refinement judgment such that they apply both in the presence and in the absence of a specification side. The rule rel-load-l' show the generalized rule rel-load-l specialized to a left refinement. We can hence continue symbolically executing the implementation up to the point where dequeue reads a value from its designated SEQ. Intuitively, by now an enqueue operation must have carried out the linearization point, *i.e.,* symbolically executed the right refinement that we placed inside the invariant (we explain how this is done below). We know this, as the enqueue obligation $\mathrm{enqueueObl}(i)$ corresponding to our dequeue operation must have been fulfilled in the invariant. And since we came first and thus were able to take the came-first token, we can conclude that the obligation contains $\models - \precsim_{id} v_s : -$. The identifier registry ensures that the id of this right refinement matches the left refinement in our goal. We take the right refinement out of the invariant in exchange for our came-first token. To "re-insert" this right refinement into our goal we use the rule rel-combine. This rule acts as a counterpart to rel-split and combines a right refinement in the context with a left refinement in the goal. After applying this rule our goal is again a standard refinement judgment, but, with a fully evaluated specification. The remaining part of proof can be completed using existing rules in ReLoC.

We now consider how the external linearization point is handled in the refinement proof of enqueue. We symbolically execute the implementation up to $\mathsf{FAA}(pushTicket, 1)$. If $pushTicket < popTicket$ then the corresponding dequeue's linearization point is external and we must step its specification forward. In the invariant this corresponds to producing a particular enqueue obligation $\mathrm{enqueueObl}(i)$. Since we do not have a came-first token for this obligation, we must produce the right refinement $\models - \precsim_{id} v_s : -$ for some id and v_s. We can do this by symbolically executing the right refinement $\models - \precsim_{id} \mathrm{dequeue}_{\mathrm{CG}}\ w : -$ present in the invariant by using a new set of rules that applies to a right refinement in one's context (rel-right-load is one such rule). From these rules the required rel-dequeue-detached can be derived.

7.1 Proof Pattern for External Linearization Points

Summarizing, the generally applicable pattern for external linearization points is as follows. One must establish an invariant that allows transferring a right refinement between the operation with an external linearization point and the operation during which the external linearization point occurs. In the refinement proof of the operation with the external linearization point, one symbolically executes the implementation up to the point where another operation may carry

[3] This treatment of the right refinement stems from the "specifications-as-resources" approach of Turon et al. [41] and is present in the model of ReLoC as well.

$$\textsc{rel-split}\quad \frac{\forall id. (\models - \lesssim_{id} e_2 : -) \mathrel{-\!\!*} \Delta \models e_1 \lesssim_{id} - : \tau}{\Delta \models e_1 \lesssim e_2 : \tau} \qquad \textsc{rel-combine}\quad \frac{\models - \lesssim_{id} e_2 : - \qquad \Delta \models e_1 \lesssim e_2 : \tau}{\Delta \models e_1 \lesssim_{id} - : \tau} \qquad \textsc{rel-load-l'}\quad \frac{\ell \hookrightarrow v \qquad (\ell \hookrightarrow v \mathrel{-\!\!*} \Delta \models K[\, v \,] \lesssim_{id} - : \tau)}{\Delta \models K[\ !\ell\ ] \lesssim_{id} - : \tau}$$

$$\textsc{rel-right-load}\quad \frac{\ell \hookrightarrow_{\mathrm{s}} v \qquad \models - \lesssim_{id} K[\ !\ell\ ] : -}{\Rrightarrow (\ell \hookrightarrow_{\mathrm{s}} v) * (\models - \lesssim_{id} K[\, v \,] : -)} \qquad \textsc{rel-dequeue-detached}\quad \frac{\mathrm{I}_{\mathrm{CG}}(w, v :: \vec{x}) \qquad (\models - \lesssim_{id} \mathsf{dequeue}_{\mathrm{CG}}\ w : -)}{\Rrightarrow \mathrm{I}_{\mathrm{CG}}(w, \vec{x}) * (\models - \lesssim_{id} v : -)}$$

Figure 8. Selected rules for external linearization points.

out the linearization point. At this point, one applies rel-split and transfers the right refinement into the invariant. Then, one uses the generalized symbolic execution rules to step the implementation forward until the point where it is certain that the external linearization point has occurred. At that point, one extracts the advanced right refinement from the invariant and applies rel-combine to merge it back into the left refinement. In the refinement of the operation during which the linearization point happens, one steps forward the implementation to the point where the external linearization point occurs, take a right refinement from the invariant, steps it forward using the symbolic execution rules for a right refinement, and puts it back into the invariant afterward.

This approach is general and in our Coq formalization we have applied it to two other examples of data structures with external linearization points: a version of the elimination-backoff stack from [18], and the red flags versus blue flags example from [42].

7.2 Changes to the ReLoC Model

We now describe how the left and right refinement judgments are defined and how the rules are encoded. The changes that we make to ReLoC rely on exposing and encapsulating a suitable amount of capabilities already present in the underlying model (described in [16]) and thus the soundness result of ReLoC is unaffected.

Recall, from [16], that the refinement judgment is defined[4] as:

$$\models e_1 \lesssim e_2 : \tau \triangleq \forall j, K.$$
$$\{\mathsf{specCtx} * j \Mapsto K[e_2]\}\ e_1\ \big\{v.\ \exists v'.\ j \Mapsto K[v'] * [\![\tau]\!](v, v')\big\}$$

That is, it is a particular Hoare triple for the left-hand side expression e_1, specifications for which talk about the thread-pool resource $j \Mapsto K[e']$ and an invariant specCtx (the latter can be ignored). These thread-pool resources are part of the ghost thread-pool: the key element in the definition of the model.

In order to obtain a right refinement, we package this thread-pool resource $j \Mapsto K[e']$ together with the invariant specCtx. The identifier for such a refinement is then a pair of the thread id j and the evaluation context K. This hides all the unnecessary details:

$$\mathsf{Id} \triangleq \{j : nat, K : ctx\}$$
$$\models - \lesssim_{id} e_2 : - \triangleq \mathsf{specCtx} * id.j \Mapsto id.K[e_2]$$

Finally, the left refinement judgment is obtained by taking the definition of a normal refinement, and stripping away the information about the right refinement from the precondition in the Hoare triple:

$$\models e_1 \lesssim_{id} - : \tau \quad \triangleq \quad \begin{array}{l} \{\mathsf{True}\} \\ \quad e_1 \\ \big\{v. \exists v', id.j \Mapsto id.K[v'] * [\![\tau]\!](v, v')\big\} \end{array} .$$

In the Coq formalization, we formalize a generalized definition that combines the left refinement $\models e_1 \lesssim_{id} - : \tau$ and the regular refinement $\models e_1 \lesssim e_2 : \tau$. This allowed us to make *tactics* that automatically apply the correct rule, depending on whether we are proving a left refinement or a regular one. Tactics allow the user to interactively carry out refinement proofs, without worrying too much about the low-level details of the rules. For example, the user can invoke a tactic `rel_load_l`, that applies either rel-load-l or rel-load-l’, depending on what is applicable. The tactics automatically determine the evaluation context K and the resource $\ell \mapsto v$ (if available).

8 Discussion: Conclusion, Related and Future Work

We now 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), ours is the first formal specification and verification of the highly-efficient and practical MPMC queue algorithm used in Meta’s Folly library. Thanks to our modular approach we also get specifications for its submodules. For example, our specification for the turn sequencer can also be used to verify other clients than the SEQ; indeed, in our Coq formalization we have used the turn sequencer to implement and verify a ticket lock.

Recently a similar bounded queue was considered by Mével and Jourdan [33]. Their motivation, approach, and challenges

[4] For reasons of clarity, the definitions given here are presented without masks and view-shifts; see [16] for details.

are different from ours. They specified the queue they considered in terms of *logically atomic triples*, while we prove contextual refinement. They verify the queue with respect to the weak memory model of multicore OCaml by using the Cosmo logic [34] , while we assume sequential consistency (a simplification compared to the C++ memory model). Their challenges stem from the complexities of weak memory, but the queue operations they verify are comparatively simpler than ours and have only fixed linearization points.

Wrt. (2), we emphasize that our extensions to ReLoC are generally applicable and suitable to support mechanized verification of a wide range of fine-grained concurrent algorithms with 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 [18], and the red flags versus blue flags example from [42].

The most closely related work not already discussed earlier in the paper is Liang and Feng's local rely/guarantee-style relational logic [30], 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 with higher-order functions (we use them to write out the constructors as closures encapsulating the internal state of the queue). Recently, a variant of Liang and Feng's logic has been formalized in Coq by Zou et. al. [49], for the purposes of verifying a concurrent file system with external linearization points. They extend the logic of Liang and Eng with abstract "helping" mechanism, which allows one thread to carry out linearization points of several other threads. It would be interesting to obtain the Coq formalization and investigate *a)* how a proof of the MPMC queue in that setting would compare with our proof in ReLoC; *b)* whether the mechanism of helpers can be implemented and applied in ReLoC.

Another relational program logic that was used for verifying algorithms with external linearization points is CaReSL [41], which also supports a functional programming language with higher-order functions and higher-order state. As mentioned, our approach to handling external linearization points is closely inspired by the "specifications-as-resources" approach of CaReSL present in the model of ReLoC.

In addition to (relational) program logics, there are many alternative methods for verifying concurrent data structures with external linearization points, including generic methods like *interval reasoning* [10, 12], or data structure specific methods like aspect-oriented proofs for concurrent queues [6]. We refer an interested reader to the survey article by Dongol and Derrick [11].

Other alternatives to contextual refinement include logically atomic Hoare triples [7, 21, 25] (which were used in the aforementioned work [33]) and HOCAP-style specifications [40], which aim at internalizing the notion of atomicity. In particular, the Iris notion of logically atomic triples is a popular correctness criterion that can handle data structures with external linearization points [25]. A logically atomic triple is a special kind of Hoare triple for a single program—unlike ReLoC's refinement judgment which relates an implementation to a specification. One strong point of logically atomic triples is that they are easy to use and build upon inside the Iris logic. On the other hand, they do not yield as strong results outside the logic as ReLoC's refinement judgment, which implies contextual refinement. Recently, logically atomic triples have been shown to imply linearizability[3], but only in a simpler first-order setting.

Contextual refinement is related to another popular correctness for concurrent algorithms: *linearizability* [19]. While there is an abundance of methods for verifying or checking linearizability, it has mainly been considered for first-order languages and with certain restrictions placed on how clients can interact with the concurrent algorithm.[5] To the best of our knowledge, linearizability has not even been properly defined for a programming language with features that we consider here (*e.g.,* higher-order functions, higher-order state, fork-based concurrency).

Acknowledgments

We are grateful to Peter O'Hearn from Meta who suggested to us that the MPMC queue would be an interesting challenge to specify and verify. We also thank Nathan Bronson and Maged Michael from Meta for informing us about the history of the MPMC queue development at Meta. 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. The second author was supported by VIDI Project No. 016.Vidi.189.046 (Unifying Correctness for Communicating Software).

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] Lars Birkedal and Aleš Bizjak. 2021. *Lecture Notes on Iris: Higher-Order Concurrent Separation Logic.* https://iris-project.org/tutorial-material.html

[3] Lars Birkedal, Thomas Dinsdale-Young, Armaël Guéneau, Guilhem Jaber, Kasper Svendsen, and Nikos Tzevelekos. 2021. Theorems for free from separation logic specifications. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–29. https://doi.org/10.1145/3473586

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

[5] 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

[5] In such a setting contextual refinement and linearizability are equivalent [13].

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

[6] Soham Chakraborty, Thomas A. Henzinger, Ali Sezgin, and Viktor Vafeiadis. 2015. Aspect-oriented linearizability proofs. *Log. Methods Comput. Sci.* 11, 1 (2015). https://doi.org/10.2168/LMCS-11(1:20)2015

[7] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A logic for time and data abstraction. In *ECOOP (LNCS, Vol. 8586)*. 207–231. https://doi.org/10.1007/978-3-662-44202-9_9

[8] 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

[9] 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, Vol. 6183)*, Theo D'Hondt (Ed.). Springer, 504–528. https://doi.org/10.1007/978-3-642-14107-2_24

[10] Brijesh Dongol and John Derrick. 2013. Simplifying proofs of linearisability using layers of abstraction. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.* 66 (2013). https://doi.org/10.14279/tuj.eceasst.66.889

[11] Brijesh Dongol and John Derrick. 2015. Verifying Linearisability: A Comparative Survey. *ACM Comput. Surv.* 48, 2 (2015), 19:1–19:43. https://doi.org/10.1145/2796550

[12] Brijesh Dongol, John Derrick, and Ian J. Hayes. 2012. Fractional Permissions and Non-Deterministic Evaluators in Interval Temporal Logic. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.* 53 (2012). https://doi.org/10.14279/tuj.eceasst.53.792

[13] 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

[14] Hubertus Franke, Rusty Russell, and Matthew Kirkwood. 2002. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. In *Proceedings of Ottawa Linux Symposium*, Vol. 85. 479–495.

[15] 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

[16] 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

[17] Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. 2007. Local Reasoning for Storable Locks and Threads. In *Programming Languages and Systems, 5th Asian Symposium, APLAS 2007, Singapore, November 29-December 1, 2007, Proceedings*. 19–37. https://doi.org/10.1007/978-3-540-76637-7_3

[18] 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

[19] 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

[20] John Hughes. 1989. Why Functional Programming Matters. *Comput. J.* 32, 2 (1989), 98–107. https://doi.org/10.1093/comjnl/32.2.98

[21] 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

[22] 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

[23] 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

[24] 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

[25] 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

[26] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain (LIPIcs, Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 17:1–17:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.17

[27] 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

[28] 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, Vol. 10201)*, Hongseok Yang (Ed.). Springer, 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

[29] 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

[30] 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

[31] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for scalable synchronization on shared-memory multiprocessors. *TOCS* 9, 1 (1991), 21–65. https://doi.org/10.1145/103727.103729

[32] Meta. 2021. Folly: Facebook Open-source Library. https://github.com/facebook/folly, last accessed: 2021-02-25.

[33] Glen Mével and Jacques-Henri Jourdan. 2021. Formal verification of a concurrent bounded queue in a weak memory model. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–29. https://doi.org/10.1145/3473571

[34] Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2020. Cosmo: a concurrent separation logic for multicore OCaml. *Proc. ACM Program. Lang.* 4, ICFP (2020), 96:1–96:29. https://doi.org/10.1145/3408978

[35] 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

[36] 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, Vol. 8410)*, Zhong Shao (Ed.). Springer, 290–310. https://doi.org/10.1007/978-3-642-54833-8_16

[37] 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

[38] 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

[39] 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, Vol. 8410)*, Zhong Shao (Ed.). Springer, 149–168. https://doi.org/10.1007/978-3-642-54833-8_9

[40] Kasper Svendsen, Lars Birkedal, and Matthew Parkinson. 2013. Modular Reasoning about Separation of Concurrent Data Structures. In *ESOP (LNCS, Vol. 7792)*. 169–188. https://doi.org/10.1007/978-3-642-37036-6_11

[41] 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

[42] 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

[43] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: navigating weak memory with ghosts, protocols, and separation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*. 691–707. https://doi.org/10.1145/2660193.2660243

[44] 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

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

[46] 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, Vol. 4703)*, Luís Caires and Vasco Thudichum Vasconcelos (Eds.). Springer, 256–271. https://doi.org/10.1007/978-3-540-74407-8_18

[47] Simon Friis Vindum and Lars Birkedal. 2021. Contextual refinement of the Michael-Scott queue (proof pearl). , 76–90 pages. https://doi.org/10.1145/3437992.3439930

[48] Simon Friis Vindum, Dan Frumin, and Lars Birkedal. 2021. *Coq development for "Mechanized Verification of a Fine-Grained Concurrent Queue from Meta's Folly Library"*. https://doi.org/10.5281/zenodo.5770802

[49] 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