Formal Verification of a Concurrent Bounded Queue in a Weak Memory Model

GLEN MÉVEL, Inria, Université Paris-Saclay, CNRS, ENS Paris-Saclay, Laboratoire Méthodes Formelles, France
JACQUES-HENRI JOURDAN, Université Paris-Saclay, CNRS, ENS Paris-Saclay, Laboratoire Méthodes Formelles, France

We use Cosmo, a modern concurrent separation logic, to formally specify and verify an implementation of a multiple-producer multiple-consumer concurrent queue in the setting of the Multicore OCaml weak memory model. We view this result as a demonstration and experimental verification of the manner in which Cosmo allows modular and formal reasoning about advanced concurrent data structures. In particular, we show how the joint use of logically atomic triples and of Cosmo's views makes it possible to describe precisely in the specification the interaction between the queue library and the weak memory model.

CCS Concepts: • **Theory of computation → Separation logic**; **Program verification**; **Program specifications**.

Additional Key Words and Phrases: separation logic, program verification, concurrency, weak memory, concurrent queue

ACM Reference Format:
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, Article 66 (August 2021), 29 pages. https://doi.org/10.1145/3473571

1 INTRODUCTION

Advances in multicore hardware during the last two decades have created a need for powerful tools for writing efficient and trustworthy multicore software. These tools include well-designed programming languages and their compilers, efficient thread-safe libraries, and expressive program logics for proving the correctness of these and of the applications that exploit them. While some such verification tools already exist, researchers are only beginning to explore whether and how these tools can be exploited to modularly specify and verify realistic libraries that support fine-grained shared-memory concurrency.

Most of the programming languages that support multicore programming present shared memory as the primitive means of communication between threads. Although this design choice offers the greatest flexibility for writing efficient programs, it comes at an important cost: in order to achieve maximal efficiency, shared-memory concurrency cannot follow an intuitive *sequentially consistent* semantics [Lamport 1979], where all threads have the same view of memory at every time. Instead, these programming languages usually come with a subtle *weak memory model*, which defines precisely how the memory accesses performed by different threads may interact

Authors' addresses: Glen Mével, Inria, Université Paris-Saclay, CNRS, ENS Paris-Saclay, Laboratoire Méthodes Formelles, 91190, Gif-sur-Yvette, France, glen.mevel@inria.fr; Jacques-Henri Jourdan, Université Paris-Saclay, CNRS, ENS Paris-Saclay, Laboratoire Méthodes Formelles, 91190, Gif-sur-Yvette, France, jacques-henri.jourdan@universite-paris-saclay.fr.

2475-1421/2021/8-ART66
https://doi.org/10.1145/3473571

with each other. This is the case, for example, of C, C++ and Rust, which share the *C11 memory model* [Batty et al. 2011; Lahav et al. 2017]; of Java and of the languages based on the JVM [Manson et al. 2005; Lochbihler 2012; Bender and Palsberg 2019]; and of the Multicore extension of the OCaml programming language [Dolan et al. 2018, 2020].

Because its semantics is subtle, shared-memory concurrency creates difficult and interesting challenges in modular program verification. Just in the past fifteen years, a large variety of concurrent separation logics have been designed in order to meet the challenge of formally specifying and verifying programs that exploit shared-memory concurrency. Brookes [2004] and O'Hearn [2007] introduced Concurrent Separation Logic [Brookes and O'Hearn 2016], which supported coarse-grain sharing of resources via mutexes. Their approach was gradually improved over the years, leading to expressive, higher-order separation logics, such as Iris [Jung et al. 2015, 2018]. Iris is able to express complex concurrent protocols, thanks to mechanisms such as *ghost state* and *invariants*, and supports reasoning about fine-grain concurrency, at the level of individual memory accesses. Concurrent data structures, such as mutexes, need not be considered primitive any more; they can be implemented and verified. Still, plain Iris is restricted to sequentially consistent semantics: it does not support weak memory models. A new generation of logics remove this restriction, for various memory models: GPS [Turon et al. 2014], iGPS [Kaiser et al. 2017] and iRC11 [Dang et al. 2020] target fragments of the C11 memory model, while Cosmo [Mével et al. 2020] targets the Multicore OCaml memory model. iGPS, iRC11 and Cosmo are based on Iris.

These logics settle a strong theoretical ground; their confirmation as practical tools, however, needs a demonstration that they allow the modular verification of realistic multicore programs. In particular, they must enable their users to precisely specify and verify concurrent data structure implementations. A concurrent queue is an archetypal example of such a data structure: it is widely used in practice, for example to manage a sequence of tasks that are generated and handled by different threads. While a coarse-grained implementation—that is, a sequential implementation protected by a lock—would certainly be correct, there exist implementations which yield better performance, especially under heavy contention, based on subtle fine-grained memory accesses. These implementations are delicate and often rely on subtle properties of the memory model. An informal correctness argument is difficult, likely unreliable, hence unconvincing. Thus, concurrent data structures are prime candidates for formal verification. In fact, many machine-checked proofs of concurrent data structures have appeared in the literature already [Parkinson et al. 2007; Frumin et al. 2018, 2020; Zakowski et al. 2018], but relatively few verification efforts take place in a weak-memory setting [Lê et al. 2013b,a], and fewer still rely on a modular methodology, where a proof of a concurrent data structure and a proof of its client (perhaps a concurrent application, or another concurrent data structure) can be modularly combined.

In this paper, we present a specification of a concurrent queue, and we formally verify that a particular implementation satisfies this specification. While other such formalizations already exist in a sequentially-consistent setting [Vindum and Birkedal 2021; Vindum et al. 2021], we consider a weak-memory setting. Such a formalization effort is innovative and challenging in several aspects.

- Weak memory models are infamous for the subtlety of the reasoning that they impose. We choose to use Cosmo [Mével et al. 2020], a recently-developed concurrent separation logic based on Iris which supports the weak memory model of Multicore OCaml [Dolan et al. 2020, 2018]. We believe that this memory model and program logic strike a good balance between the ease of reasoning enabled by the logic and the flexibility and performance allowed by the memory model.
- In our quest for applicability, we wish to propose a realistic queue implementation, which could be used in real-world programs. Since Multicore OCaml is still at an experimental stage

and does not offer a wide variety of concurrent data structures yet, its ecosystem may benefit from this new library. We take inspiration from a well-established algorithm [Rigtorp 2021] that has been used in production in several applications.

- The specification of the concurrent queue should indicate that it behaves as if all of its operations acted atomically on a common shared state, even though in reality they access distinct parts of the memory and require many machine instructions. To address this challenge, we use the recently-developed concept of *logical atomicity* [Jung et al. 2015; Jung 2019; da Rocha Pinto et al. 2014], which we transport to the setting of Cosmo.
- To the best of our knowledge, this is the first use of logical atomicity in a weak-memory setting. This raises new questions: for instance, even though our implementation realizes a total order on the operations on a queue, it offers strictly weaker guarantees than would be offered by a coarse-grained implementation. Indeed, in the context of a weak memory model, the specification of a concurrent data structure must describe not only the result of its operations, but also the manner in which these are synchronized, that is, the *happens-before* relationships that exist between these. This additional information allows reasoning about accesses to areas of memory *outside* of the data structure itself. This is crucial, for example, if a queue is used to transfer the ownership of a piece of memory from a producer to a consumer: there must exist a happens-before relationship between the enqueue operation and the corresponding dequeue operation, so as to ensure that the consumer acquires the producer's view of this piece of memory. Our specification faithfully captures a subtle behavior of the implementation: even though operations are totally ordered by logical atomicity, not all operations are ordered by happens-before—but *some* are.
- We use the Coq proof assistant [The Coq development team 2020] to formally verify our proofs. Our development is available from our repository [Mével et al. 2021].

We believe our approach, whose key ingredients are Cosmo and logical atomicity, scales to other memory models and other data structures. Indeed, first, the core of Cosmo (beyond basic Separation Logic) is a logic for reasoning with *views*, an operational description of the memory model; other memory models than that of Multicore OCaml can also be termed in this fashion, as iGPS [Kaiser et al. 2017] and iRC11 [Dang et al. 2020] have demonstrated for C11. Second, logical atomicity has already successfully been used for various data structures in the Iris community [Iris developers and contributors 2021; Frumin et al. 2020].

The paper begins with a detailed explanation of the specification of a concurrent queue (§2). Then, we present an implementation of the queue in Multicore OCaml (§3), and explain our proof of its correctness (§4). Next, we demonstrate that our specification is indeed usable, by exploiting it in the context of a simple piece of client code, where the concurrent queue is used to establish a pipeline between a set of producers and a set of consumers (§5). The paper ends with a review of the related work (§6).

2 SPECIFICATION OF A MPMC QUEUE IN A CONCURRENT SEPARATION LOGIC

A queue is a first-in first-out container data structure. At any time, it holds an ordered list of items. It supports two main operations: enqueue inserts an item at one extremity of the queue (the *head*); dequeue extracts an item—if there is one—from the other extremity (the *tail*).

In a concurrent setting, a legitimate question is whether several threads can operate the queue safely. The answer depends on the implementation. Possible restrictions include allowing only one thread to enqueue (single producer), or allowing only one thread to dequeue (single consumer), or both. In this paper we study an implementation of a *multiple-producer, multiple-consumer* (MPMC) queue, that is, a queue in which any number of threads are allowed to enqueue and dequeue.

$$\{\mathsf{True}\}\ \ \mathtt{make}\ ()\ \ \{\lambda q.\ \mathsf{IsQueue}\ q\ []\}$$

$$\{\mathsf{IsQueue}\ q\ [v_0, \dots, v_{n-1}]\}\ \ \mathtt{enqueue}\ q\ v\ \ \{\lambda ().\ \mathsf{IsQueue}\ q\ [v_0, \dots, v_{n-1}, v]\}$$

$$\{\mathsf{IsQueue}\ q\ [v_0, \dots, v_{n-1}]\}\ \ \mathtt{dequeue}\ q\ \ \{\lambda v.\ 1 \le n \ * \ v = v_0 \ * \ \mathsf{IsQueue}\ q\ [v_1, \dots, v_{n-1}]\}$$

Fig. 1. A specification of the "queue" data structure in a sequential setting

In addition, this queue is *bounded*, that is, it occupies no more than a fixed memory size. A motivation for that trait is that items may be enqueued more frequently than they are dequeued; in this situation, a bounded queue has no risk of exhausting system memory; instead, if the maximum size is reached, enqueuing either blocks or fails.

2.1 Specification in a Sequential Setting

Let us start by assuming a sequential setting. We can then use standard Separation Logic [Reynolds 2002] to reason about programs and the resources they manipulate. In Separation Logic, a queue q holding n items $[v_0, \dots, v_{n-1}]$, where the left extremity of the list is the tail and the right extremity is the head, can be represented with an assertion:

$$\mathsf{IsQueue}\ q\ [v_0, \dots, v_{n-1}]$$

As is usual in Separation Logic, this *representation predicate* asserts the unique ownership of the entire data structure. When holding it, we can safely manipulate the queue without risk of invalidating other assertions about resources that may alias parts of our queue. In particular, the representation predicate cannot be duplicated; we say that it is *exclusive*.

The operations of the queue admit a simple sequential specification which is presented in Figure 1. We use the standard Hoare triple notation, with a partial correctness meaning. The asterisk ($*$) is the separating conjunction.

- The function make has no prerequisite and gives us the ownership of a new empty queue.
- If we own a queue q, then we can enqueue some item v into q; if this operation ever returns, then it must return the unit value () and give us back the ownership of q, where v has been appended at the head.
- Conversely, if we own a queue q, then we can dequeue from q; if this operation ever returns, then it must return the first item v_0 found at the tail of q, and it gives us back the ownership of q, where that first item has been removed.

This specification implies that dequeue cannot possibly return when the queue is empty ($n = 0$); in this case, it must loop forever. This is pointless in a sequential setting, but becomes meaningful in the presence of concurrency, where it makes sense for dequeue to wait until an item is enqueued. This specification also applies to bounded queues, where (somewhat analogously) enqueue loops when the capacity is reached ($n = C$), waiting until room becomes available.

2.2 Specification under Sequential Consistency

We now consider a situation where several threads can access the queue concurrently. Let us assume for now that the memory model is sequentially consistent [Lamport 1979]. In a seminal

$$\frac{}{\mathsf{persistent}(\mathsf{QueueInv}\ q\ \gamma)}$$

$$\begin{array}{l}\left\{\mathsf{True}\right\}\\ \quad \mathsf{make}\ ()\\ \left\{\lambda q.\ \exists \gamma.\ \mathsf{QueueInv}\ q\ \gamma \ \ast\ \mathsf{IsQueue}\ \gamma\ []\right\}\end{array}$$

$$\frac{\mathsf{QueueInv}\ q\ \gamma}{\begin{array}{l}\left\langle n, v_0, \dots, v_{n-1}.\ \mathsf{IsQueue}\ \gamma\ [v_0, \dots, v_{n-1}]\right\rangle\\ \quad \mathsf{enqueue}\ q\ v\\ \left\langle \lambda ().\ \mathsf{IsQueue}\ \gamma\ [v_0, \dots, v_{n-1}, v]\right\rangle\end{array}}$$

$$\frac{\mathsf{QueueInv}\ q\ \gamma}{\begin{array}{l}\left\langle n, v_0, \dots, v_{n-1}.\ \mathsf{IsQueue}\ \gamma\ [v_0, \dots, v_{n-1}]\right\rangle\\ \quad \mathsf{dequeue}\ q\\ \left\langle \lambda v.\ 1 \leq n \ \ast\ v = v_0 \ \ast\ \mathsf{IsQueue}\ \gamma\ [v_1, \dots, v_{n-1}]\right\rangle\end{array}}$$

Fig. 2. A specification of the "queue" data structure in a sequentially consistent memory model

work, Brookes [2004] and O'Hearn [2007] devised Concurrent Separation Logic, an extension of Separation Logic which enables to reason about programs where several threads can access a same piece of memory. Though the original logic achieves sharing through hard-wired "conditional critical regions", it has spawned a variety of descendants lifting this limitation and pushing further the applicability of such separation logics. In this work, we use Iris [Jung et al. 2018], a modular framework for building separation logics.

In a derivative of Concurrent Separation Logic, we may retain the exact same specification as is presented in Figure 1, recalling that IsQueue $q\ [v_0, \dots, v_{n-1}]$ is an exclusive assertion: a thread that has this assertion can safely assume to be the only one allowed to operate on the queue. A client application can transfer this assertion between threads via some means of synchronization: for instance, it may use a lock to guard all operations on the shared queue, following the approach of Concurrent Separation Logic. However this coarse grain concurrency has a run-time penalty, and it also creates some contention on the use of the queue. These costs are often unnecessary, as many data structures are designed specifically to support concurrent accesses. In this paper, as stated, we wish to prove the correctness of a MPMC queue implementation, which should thus ensure, by itself, thread safety. Hence we can achieve finer-grain concurrency, where operating on a queue does not require its exclusive ownership.

In this context, another option is for the client to share this ownership among several threads, logically. In an Iris-like logic, one would typically place the exclusive ownership in an *invariant.* An invariant is an assertion which is agreed upon by all threads, and is owned by anyone; it remains true forever. As the public state of the queue—the list $[v_0, \dots, v_{n-1}]$ of currently stored items—would only be known from that invariant, the client would also express in there the properties about this state that their particular application needs. Then, when one of the threads needs to access the shared resource, it can *open* the invariant, get the assertions it contains, perform the desired operation on the shared state, reestablish the invariant, and finally close it. However, to ensure soundness in the face of concurrency, the use of invariants in Iris obeys a strict constraint: they can remain open during at most one step of execution. Unfortunately, enqueue and dequeue are complex operations which, *a priori*, take several steps. Hence a client would be unable to open their invariant around the triples shown in Figure 1. Yet these operations are "atomic" in some empirical sense.

The concept of logical atomicity [Jacobs and Piessens 2011; Jung 2019; Jung et al. 2015, §7] aims at addressing that difficulty. To use it, we substitute ordinary Hoare triples with *logically atomic*

$$\textsc{LAHoare}\quad \frac{\langle x.\, P\rangle\ e\ \langle Q\rangle}{\forall x.\ \{P\}\ e\ \{Q\}} \qquad\qquad \textsc{LAInv}\quad \frac{\langle x.\ \triangleright I * P\rangle\ e\ \langle \triangleright I * Q\rangle}{\boxed{I} \vdash \langle x.\, P\rangle\ e\ \langle Q\rangle}$$

Fig. 3. Selected rules for logically atomic triples

triples. Two important reasoning rules for logically atomic triples are given in Figure 3.[1] A logically atomic triple is denoted with angle brackets $\langle \ldots \rangle$. Just like an ordinary triple, it specifies a program fragment with a precondition and a postcondition. In fact, as witnessed by rule LAHoare, one can deduce an ordinary Hoare triple from a logically atomic triple. The core difference is that, thanks to rule LAInv, invariants can be opened around a logically atomic triple, regardless of the number of execution steps of the program fragment: in a sense, when a function is specified using a logically atomic triple, one states that said function behaves as if it were atomic. The definition of logically atomic triples is further discussed in §4.6 and given with detail in previous work [Jung 2019; Jung et al. 2015, §7]. We now try to give an intuition of that concept: a logically atomic triple $\langle P\rangle\ e\ \langle Q\rangle$ states, roughly, that the expression e contains an atomic instruction, called the *commit point*, which has P as a precondition and Q as a postcondition. Because it is atomic, invariants can be opened around that commit point.

Using logically atomic triples, the specification can be written as shown in Figure 2. It closely resembles that of the sequential setting (Figure 1). The first noticeable difference is the use of angle brackets $\langle \ldots \rangle$ denoting logically atomic triples instead of curly brackets $\{\ldots\}$ for ordinary Hoare triples.

Another difference is the presence, in the syntax of logically atomic triples, of an explicit binder for some variables $(n, v_0, \ldots, v_{n-1})$. This binder addresses a subtlety of logical atomicity: a client calling enqueue or dequeue does not know in advance the state of the queue at the commit point, which is when the precondition and postcondition are to be interpreted. Hence, both formulas have to be parameterized by said shared state. Said otherwise, a logically atomic triple provides a *family* of pre/postcondition pairs covering every possible shared state at the commit point.

The last departure from the sequential specification is that the representation predicate is split into two parts: a persistent[2] assertion $\mathsf{QueueInv}\ q\ \gamma$ and an exclusive assertion $\mathsf{IsQueue}\ \gamma\ [v_0, \ldots, v_{n-1}]$, connected by a ghost name[3] γ. That splitting is an artifact of our correctness proof technique, which we detail in §4. Note that this does not complicate the use of the queue by the client: both assertions are produced when creating the queue, and while the exclusive component can be put in an invariant as before, the persistent component can be directly duplicated and distributed to all threads.[4]

The use of such a specification in a concrete example will be detailed in §5.3. For now, we illustrate how a weaker specification can be easily deduced from this one.

[1]Following Iris notations, $\boxed{I}$ is an invariant whose content is the assertion I, and $\triangleright$ is a step-indexing modality, a technicality of Iris that we can ignore in this paper.

[2]In Iris terms, *persistent* qualifies an assertion that is duplicable. Once established, such an assertion holds forever.

[3]Ghost names in Iris are identifiers for pieces of ghost state. We say more on this in §4.

[4]An Iris expert may want to conceal the queue invariant, $\mathsf{QueueInv}\ q\ \gamma$, inside $\mathsf{IsQueue}\ \gamma\ [v_0, \ldots, v_{n-1}]$. However, we need to access this invariant at various places other than the commit point. This is feasible with a more elaborate definition of logically atomic triples than the one given in this paper, so that they support *aborting* [Jung 2019]. Another drawback is that we would lose the timelessness of the representation predicate.

A persistent specification. If it were not for logical atomicity, and we still wanted to share the ownership of the queue, we would have little choice left than renouncing to an exclusive representation predicate. Only a persistent assertion would be provided, because the description of the public state has to be stable in the face of interference by other threads. The resulting specification would be much weaker. For example, we may merely specify that all of the elements stored in the queue satisfy some predicate Φ. In doing so, we lose most structural properties of a queue: the same specification could also describe a stack or a bag.

$$\frac{}{\mathsf{persistent}(\mathsf{QueuePersistent}\ q\ \Phi)} \qquad \frac{\mathsf{QueuePersistent}\ q\ \Phi}{\{\Phi\, v\}\ \mathsf{enqueue}\ q\ v\ \{\lambda().\,\mathsf{True}\}} \qquad \frac{\mathsf{QueuePersistent}\ q\ \Phi}{\{\mathsf{True}\}\ \mathsf{dequeue}\ q\ \{\lambda v.\,\Phi\, v\}}$$

To derive these Hoare triples from the ones of Figure 2, one simply defines the persistent assertion as follows, where the boxed assertion is an Iris invariant:

$$\mathsf{QueuePersistent}\ q\ \Phi \ \triangleq\ \exists\gamma.\ \ast\begin{cases}\mathsf{QueueInv}\ q\ \gamma \\ \boxed{\exists n, v_0, \ldots, v_{n-1}.\ \mathsf{IsQueue}\ \gamma\ [v_0, \ldots, v_{n-1}]\ \ast\ \Phi\, v_0\ \ast\ \cdots\ \ast\ \Phi\, v_{n-1}}\end{cases}$$

This assertion is trivial to produce at the creation of the queue, when `make` hands us the assertions QueueInv q γ and IsQueue γ []. Then, for proving the weaker specification of enqueue and dequeue, one opens the invariant around the associated logically atomic triples.

2.3 Specification under Weak Memory

Up to now, we have ignored the weakly consistent behavior of the semantics of Multicore OCaml. In this section, we take this aspect into account and propose a refined specification.

We use the separation logic Cosmo [Mével et al. 2020], which provides a proof framework for the weak memory model of Multicore OCaml. This memory model has been described by Dolan et al. [2018] under the form of a small-step operational semantics. Following previous authors, such as Kaiser et al. [2017], the semantics features a notion of *view*[5] which captures the essence of weak memory, namely, the fact that each thread has different knowledge of the state of the shared memory. A view represents this subjective knowledge. Cosmo builds on this operational semantics, and brings views to the level of the program logic. All assertions of the logic depend on an *ambient view*, which corresponds to the current view of the subject thread. Thus, Cosmo assertions are in general *subjective*, that is, implicitly dependent on an ambient view.

Because Cosmo is based on Iris, logically atomic triples can also be defined in Cosmo. In fact, the specification shown in Figure 2 still applies.

Yet, as such, it is of little value in a weakly consistent context. Indeed, as explained in §2.2, it is designed so that IsQueue γ $[v_0, \ldots, v_{n-1}]$ can be shared among threads by means of an invariant. But, in Cosmo, invariants are restricted to a special class of assertions called *objective* assertions—those assertions that do not actually depend on the ambient view. Hence, our first addition to the specification is to stipulate that the representation predicate is objective. This reflects the fact that there exists a total order on the updates to the logical state, on which all threads objectively agree.

Even with this addition, the specification given in §2.2 is not precise enough to verify interesting clients such as the one described in §5. Indeed, in a weakly consistent setting, one typically expects a concurrent data structure to establish synchronization between some of its concurrent accesses. For example, imagine that thread A enqueues a pointer to a complex data structure (say, a hash table). Then, when thread B dequeues this pointer, B should obtain the unique ownership of the hash table and be able to access it accordingly. In a weakly consistent memory model, B expects to see all of the changes that A has made to the data structure. This is guaranteed only if there is a *happens-before* relationship from the enqueuing event to the dequeuing event.

[5]Dolan et al. [2018] call these views *frontiers*.

Seen-Zero	Seen-Two	Split-Subjective-Objective
$\vdash\ \uparrow\bot$	$\uparrow\mathcal{V}_1 * \uparrow\mathcal{V}_2 \dashv\vdash \uparrow(\mathcal{V}_1 \sqcup \mathcal{V}_2)$	$P \dashv\vdash \exists\mathcal{V}.\,(\uparrow\mathcal{V} * P @ \mathcal{V})$

Fig. 4. Selected Cosmo rules

$$\frac{}{\mathsf{persistent}(\mathsf{QueueInv}\,q\,\gamma)} \qquad \frac{}{\mathsf{objective}(\mathsf{IsQueue}\,\gamma\,\mathcal{T}\,\mathcal{H}\,[(v_0,\mathcal{V}_0),\ldots,(v_{n-1},\mathcal{V}_{n-1})])}$$

$$\begin{array}{l}\left\{\uparrow\mathcal{V}_0\right\}\\ \quad\texttt{make ()}\\ \left\{\lambda q.\,\exists\gamma.\,\mathsf{QueueInv}\,q\,\gamma \;*\; \mathsf{IsQueue}\,\gamma\,\mathcal{V}_0\,\mathcal{V}_0\,[\,]\right\}\end{array}$$

$$\frac{\mathsf{QueueInv}\,q\,\gamma}{\begin{array}{l}\left\langle \begin{array}{l}\mathcal{T},\mathcal{H},n,(v_0,\mathcal{V}_0),\ldots,(v_{n-1},\mathcal{V}_{n-1}).\\ \quad\mathsf{IsQueue}\,\gamma\,\mathcal{T}\;\mathcal{H}\;[(v_0,\mathcal{V}_0),\ldots,(v_{n-1},\mathcal{V}_{n-1})] \;*\; \uparrow\mathcal{V}\end{array}\right\rangle\\ \quad\texttt{enqueue}\ q\ v\\ \left\langle \lambda().\,\mathsf{IsQueue}\,\gamma\,\mathcal{T}\,(\mathcal{H}\sqcup\mathcal{V})\,[(v_0,\mathcal{V}_0),\ldots,(v_{n-1},\mathcal{V}_{n-1}),(v,\mathcal{V})] \;*\; \uparrow\mathcal{H}\right\rangle\end{array}}$$

$$\frac{\mathsf{QueueInv}\,q\,\gamma}{\begin{array}{l}\left\langle \begin{array}{l}\mathcal{T},\mathcal{H},n,(v_0,\mathcal{V}_0),\ldots,(v_{n-1},\mathcal{V}_{n-1}).\\ \quad\mathsf{IsQueue}\,\gamma\;\mathcal{T}\;\mathcal{H}\,[(v_0,\mathcal{V}_0),\ldots,(v_{n-1},\mathcal{V}_{n-1})] \;*\; \uparrow\mathcal{V}\end{array}\right\rangle\\ \quad\texttt{dequeue}\ q\\ \left\langle \lambda v.\,\mathsf{IsQueue}\,\gamma\,(\mathcal{T}\sqcup\mathcal{V})\,\mathcal{H}\,[(v_1,\mathcal{V}_1),\ldots,(v_{n-1},\mathcal{V}_{n-1})] \;*\; \uparrow\mathcal{T} \;*\; \uparrow\mathcal{V}_0 \;*\; 1\leq n \;*\; v=v_0\right\rangle\end{array}}$$

$$\frac{\mathsf{QueueInv}\,q\,\gamma}{\begin{array}{l}\left\langle \begin{array}{l}\mathcal{T},\mathcal{H},n,(v_0,\mathcal{V}_0),\ldots,(v_{n-1},\mathcal{V}_{n-1}).\\ \quad\mathsf{IsQueue}\,\gamma\,\mathcal{T}\;\mathcal{H}\;[(v_0,\mathcal{V}_0),\ldots,(v_{n-1},\mathcal{V}_{n-1})] \;*\; \uparrow\mathcal{V}\end{array}\right\rangle\\ \quad\texttt{try\_enqueue}\ q\ v\\ \left\langle \lambda b.\,\bigvee\left[\begin{array}{l}\mathsf{IsQueue}\,\gamma\,\mathcal{T}\;\mathcal{H}\;[(v_0,\mathcal{V}_0),\ldots,(v_{n-1},\mathcal{V}_{n-1})] \;*\; b=\texttt{false}\\ \mathsf{IsQueue}\,\gamma\,\mathcal{T}\,(\mathcal{H}\sqcup\mathcal{V})\,[(v_0,\mathcal{V}_0),\ldots,(v_{n-1},\mathcal{V}_{n-1}),(v,\mathcal{V})] \;*\; \uparrow\mathcal{H} \;*\; b=\texttt{true}\end{array}\right.\right\rangle\end{array}}$$

$$\frac{\mathsf{QueueInv}\,q\,\gamma}{\begin{array}{l}\left\langle \begin{array}{l}\mathcal{T},\mathcal{H},n,(v_0,\mathcal{V}_0),\ldots,(v_{n-1},\mathcal{V}_{n-1}).\\ \quad\mathsf{IsQueue}\,\gamma\;\mathcal{T}\;\mathcal{H}\,[(v_0,\mathcal{V}_0),\ldots,(v_{n-1},\mathcal{V}_{n-1})] \;*\; \uparrow\mathcal{V}\end{array}\right\rangle\\ \quad\texttt{try\_dequeue}\ q\\ \left\langle \lambda v^?.\,\bigvee\left[\begin{array}{l}\mathsf{IsQueue}\,\gamma\;\mathcal{T}\;\mathcal{H}\,[(v_0,\mathcal{V}_0),\ldots,(v_{n-1},\mathcal{V}_{n-1})] \;*\; v^?=\mathsf{None}\\ \mathsf{IsQueue}\,\gamma\,(\mathcal{T}\sqcup\mathcal{V})\,\mathcal{H}\,[(v_1,\mathcal{V}_1),\ldots,(v_{n-1},\mathcal{V}_{n-1})] \;*\; \uparrow\mathcal{T} \;*\; \uparrow\mathcal{V}_0 \;*\; 1\leq n \;*\; v^?=\mathsf{Some}\,v_0\end{array}\right.\right\rangle\end{array}}$$

Fig. 5. A specification of the “queue” data structure in a weak memory model

One possibility would be to guarantee that our concurrent queue implementation behaves like its coarse-grained alternative, that is, a sequential implementation guarded by a lock. This would correspond to an intuitive definition of linearizability, even though this notion is difficult to define precisely outside of the world of sequential consistency [Smith et al. 2019]. However, our concurrent queue library is weaker than that: it does guarantee *some* happens-before relationships, but not between all pairs of accesses. Namely, it guarantees a happens-before relationship:

(1) from an enqueuer to the dequeuer that obtains the corresponding item;
(2) from an enqueuer to the following enqueuers;
(3) from a dequeuer to the following dequeuers.

The first one permits resource transfer through the queue as described in the example above.

In Cosmo, happens-before relationships can be expressed as *transfers of views*. To the user of this logic, views (denoted in this paper by calligraphic capital letters, such as $\mathcal{T}, \mathcal{H}, \mathcal{V}, \mathcal{S}$) are abstract values equipped with lattice structure: the least view $\bot$ is the empty view, and larger views correspond to more recent knowledge. Cosmo features two kinds of assertions to deal with views. First, if $\mathcal{V}$ is a view, then the persistent assertion $\uparrow \mathcal{V}$ indicates that the current thread has the knowledge contained in $\mathcal{V}$. Second, if $\mathcal{V}$ is a view and P is a (subjective) assertion, then $P \mathbin{@} \mathcal{V}$ denotes the assertion P where $\mathcal{V}$ has been substituted for the ambient view; as it does not depend on the ambient view anymore, $P \mathbin{@} \mathcal{V}$ is objective.[6] Cosmo provides reasoning rules about these assertions, shown in Figure 4. We comment on rule Split-Subjective-Objective in the next paragraph.

In Cosmo, the usual way of specifying a happens-before relationship between two program points is by giving the client the ability to transfer any assertion of the form $\uparrow \mathcal{V}$ between these two points: this corresponds to saying that the destination program point has all the knowledge the source program point had about the shared memory. As seen later with the example of the pipeline (§5.3), this is sufficient for transferring any subjective resource from a sender to a receiver. Indeed, rule Split-Subjective-Objective says that we can split any subjective assertion P into two assertions $\uparrow \mathcal{V}$ and $P \mathbin{@} \mathcal{V}$ for some view $\mathcal{V}$ and, conversely, we can reconstruct P from two such assertions. The second part being objective, it can be shared in an invariant. Hence, to transfer P, it is enough to transfer its subjective part $\uparrow \mathcal{V}$.

In the specification of the queue, to express the happens-before relationships mentioned earlier, the representation predicate now takes more parameters:

$$\mathsf{IsQueue}\ \gamma\ \mathcal{T}\ \mathcal{H}\ [(v_0, \mathcal{V}_0), \dots, (v_{n-1}, \mathcal{V}_{n-1})]$$

(1) For each item v_k in the queue, we now have a corresponding view $\mathcal{V}_k$. This view materializes the flow of memory knowledge from the thread which enqueued the item, to the one which will dequeue it.
(2) The *head* view $\mathcal{H}$ materializes memory knowledge accumulated by successive enqueuers.
(3) The *tail* view $\mathcal{T}$ materializes memory knowledge accumulated by successive dequeuers.

The queue that we study, however, does not guarantee any happens-before relationship from a dequeuer to an enqueuer.[7] Hence, it provides fewer guarantees than a sequential queue guarded by a lock.

Interestingly, Cosmo is able to express this subtle difference between the behavior of our library and that of a lock-based implementation: the full specification under weak memory is shown

[6] An equivalent definition is the following: asserting $P \mathbin{@} \mathcal{V}$ is asserting that, even without other knowledge, if we have the knowledge contained in $\mathcal{V}$, then P holds. In other words, $P \mathbin{@} \mathcal{V}$ is objective and entails $\uparrow \mathcal{V} \twoheadrightarrow P$.

[7] This is not entirely true: the implementation shown in §3 does create a happens-before relationship from the dequeuer of rank k to the enqueuer of rank $k + C$ (hence also to all enqueuers of subsequent ranks). We choose to not reveal this in the specification, since the constant C is an implementation detail.

$x \in \textsc{Var}$	— variables
$\ell \in \textsc{Loc}$	— memory locations
$b \in \{\texttt{false}, \texttt{true}\}$	— Boolean values
$n \in \mathbb{Z}$	— integer values
$v ::= \mu f.\, \lambda x.\, e \mid \ell \mid () \mid b \mid n$	— values
$\quad \mid \texttt{None} \mid \texttt{Some}\ v \mid (v, \ldots, v)$	
$\odot \in \{\wedge, \vee, +, \times, \texttt{mod}, \ldots\}$	— operators
$e ::=$	— expressions:
$\quad x \mid \mu f.\, \lambda x.\, e \mid e\ e \mid e \odot e$	– λ-calculus with recursive functions
$\quad \mid \ell \mid () \mid b \mid n \mid \texttt{None} \mid \texttt{Some}\ e \mid (e, \ldots, e)$	– primitive values and constructors
$\quad \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$	– conditional
$\quad \mid \texttt{match}\ e\ \texttt{with Some}\ x \rightarrow e \mid \texttt{None} \rightarrow e$	– pattern matching
$\quad \mid \texttt{let}\ (x, \ldots, x) = e\ \texttt{in}\ e$	– tuples
$\quad \mid \texttt{fork}\ e$	– spawning a new thread
$\quad \mid \texttt{array}_{\mathrm{na}}[e]\ e \mid e[e]_{\mathrm{na}} \mid e[e]_{\mathrm{na}} \leftarrow e$	– non-atomic cells
$\quad \mid \texttt{array}_{\mathrm{at}}[e]\ e \mid e[e]_{\mathrm{at}} \mid e[e]_{\mathrm{at}} \leftarrow e$	– atomic cells
$\quad \mid \texttt{CAS}\ e[e]\ e\ e$	– atomic compare-and-set
$e ::= \ldots$	— syntactic sugar:
$\quad \mid \texttt{let rec}\ x\ x\ \ldots\ x = e\ \texttt{in}\ e$	– definitions of recursive functions
$\quad \mid \texttt{let}\ x\ \ldots\ x = e\ \texttt{in}\ e \mid e\,;e$	– definitions, sequencing
$\quad \mid \texttt{for}\ x\ \texttt{from}\ e\ \texttt{to}\ e\ \texttt{do}\ e\ \texttt{done}$	– loops
$\quad \mid \texttt{ref}_{\mathrm{na}}\ e \mid !_{\mathrm{na}}\, e \mid e :=_{\mathrm{na}} e$	– non-atomic references
$\quad \mid \texttt{ref}_{\mathrm{at}}\ e \mid !_{\mathrm{at}}\, e \mid e :=_{\mathrm{at}} e \mid \texttt{CAS}\ e\ e\ e$	– atomic references

Fig. 6. The syntax of a fragment of Multicore OCaml

in Figure 5. This specification extends the previous one (Figure 2) with views. The mentioned happens-before relationships are captured as follows.

(1) When a thread with a local view $\mathcal{V}$ (in other words, with $\uparrow\mathcal{V}$ as a precondition) enqueues an item v, it pairs it with the view $\mathcal{V}$. Afterwards, when another thread dequeues that same item v_0, it merges the view $\mathcal{V}_0$ that was paired with it into its own local view (in other words, it obtains $\uparrow\mathcal{V}_0$ as a postcondition).
(2) When a thread enqueues an item, it also obtains the head view $\mathcal{H}$ left by the previous enqueuer (in other words, it obtains $\uparrow\mathcal{H}$ as a postcondition), and it adds its own view $\mathcal{V}$ to the head view (which becomes $\mathcal{H} \sqcup \mathcal{V}$).
(3) When a thread dequeues an item, it also obtains the tail view $\mathcal{T}$ left by the previous dequeuer (in other words, it obtains $\uparrow\mathcal{T}$ as a postcondition), and it adds its own view $\mathcal{V}$ to the tail view (which becomes $\mathcal{T} \sqcup \mathcal{V}$).

3 IMPLEMENTATION OF A BOUNDED MPMC QUEUE USING A RING BUFFER

We now present an implementation of a bounded MPMC queue, that satisfies the specification devised in §2. For the purpose of the formalization, it is written in an idealized version of the Multicore OCaml language which we introduce in §3.1, before showing the code (§3.2, §3.3) and giving intuitions about its mode of operation (§3.4, §3.5).

$$\begin{array}{l} \{\ell[k] \rightsquigarrow_{\mathrm{na}} v\} \\ \quad \ell[k]_{\mathrm{na}} \\ \{\lambda v'.\, v' = v \;*\; \ell[k] \rightsquigarrow_{\mathrm{na}} v\} \end{array} \qquad \begin{array}{l} \{\ell[k] \rightsquigarrow_{\mathrm{na}} \_\} \\ \quad \ell[k]_{\mathrm{na}} \leftarrow v \\ \{\lambda ().\, \ell[k] \rightsquigarrow_{\mathrm{na}} v\} \end{array}$$

$$\begin{array}{l} \{\ell[k] \rightsquigarrow_{\mathrm{at}} \langle v, \mathcal{V} \rangle\} \\ \quad \ell[k]_{\mathrm{at}} \\ \{\lambda v'.\, v' = v \;*\; \ell[k] \rightsquigarrow_{\mathrm{at}} \langle v, \mathcal{V} \rangle \;*\; \uparrow \mathcal{V}\} \end{array} \qquad \begin{array}{l} \{\ell[k] \rightsquigarrow_{\mathrm{at}} \langle \_, \mathcal{V} \rangle \;*\; \uparrow \mathcal{V}'\} \\ \quad \ell[k]_{\mathrm{at}} \leftarrow v \\ \{\lambda ().\, \ell[k] \rightsquigarrow_{\mathrm{at}} \langle v, \mathcal{V} \sqcup \mathcal{V}' \rangle\} \end{array}$$

$$\begin{array}{l} \{v_0 \neq v_1 \;*\; \ell[k] \rightsquigarrow_{\mathrm{at}} \langle v_0, \mathcal{V} \rangle\} \\ \quad \mathtt{CAS}\ \ell[k]\ v_1\ v_2 \\ \left\{\lambda v'.\, \mathop{\Huge *}\left\{\begin{array}{l} v' = \mathtt{false} \\ \ell[k] \rightsquigarrow_{\mathrm{at}} \langle v_0, \mathcal{V} \rangle \\ \uparrow \mathcal{V} \end{array}\right.\right\} \end{array} \qquad \begin{array}{l} \{\ell[k] \rightsquigarrow_{\mathrm{at}} \langle v_1, \mathcal{V} \rangle \;*\; \uparrow \mathcal{V}'\} \\ \quad \mathtt{CAS}\ \ell[k]\ v_1\ v_2 \\ \left\{\lambda v'.\, \mathop{\Huge *}\left\{\begin{array}{l} v' = \mathtt{true} \\ \ell[k] \rightsquigarrow_{\mathrm{at}} \langle v_2, \mathcal{V} \sqcup \mathcal{V}' \rangle \\ \uparrow \mathcal{V} \end{array}\right.\right\} \end{array}$$

Fig. 7. Simplified Cosmo triples for the memory operations of Multicore OCaml

3.1 Multicore OCaml

The syntax of our idealized version of Multicore OCaml is presented in Figure 6. It is untyped (contrary to the actual Multicore OCaml language) and equipped with a standard call-by-value, left-to-right evaluation. The parts of interest are the memory operations. Their semantics have been described by Mével et al. [2020]; however, in the present paper, each location stores an *array* of values, whose cells act as independent memory cells with respect to the memory model. Cells are rigidly ascribed an access mode α, which is either "atomic" ($\alpha = \mathrm{at}$) or "non-atomic" ($\alpha = \mathrm{na}$). When $0 \leq n$, the expression $\mathtt{array}_\alpha[n]\ v$ allocates a block of n cells whose access mode is α and whose initial value is v. Reading from a cell with access mode α at offset k of location ℓ is written $\ell[k]_\alpha$. Writing a value v to a cell with access mode α at offset k of location ℓ is written $\ell[k]_\alpha \leftarrow v$. In addition, atomic cells support the usual compare-and-set operation: $\mathtt{CAS}\ \ell[k]\ v_1\ v_2$ reads the atomic cell at offset k of location ℓ, tests whether its value is equal to v_1, overwrites it with v_2 if that is the case, and returns the Boolean result of the test; importantly, the read and the write operations happen *atomically*. There is syntactic sugar for single-cell locations, or "references": $\mathtt{ref}_\alpha\ v$ allocates a location of length one, $!_\alpha\ \ell$ reads at offset zero, $\ell :=_\alpha v$ writes at offset zero and $\mathtt{CAS}\ \ell\ v_1\ v_2$ performs compare-and-set at offset zero.

Lastly, `fork` e creates a new thread which executes e. The expression `fork` e returns the unit value `()` without waiting for the completion of the new thread.

The memory model of Multicore OCaml describes how the memory operations interact with the shared memory. Dolan et al. [2018] give an operational account, which is out of the scope of the current paper. However, to give some intuition, we show in Figure 7 what this semantics translates to in the Cosmo program logic. In the following paragraphs, we give an overview of these triples which are explained more thoroughly by Mével et al. [2020].

There are different points-to assertions for atomic cells and for non-atomic cells. As is standard, these predicates assert unique ownership of the cells.

Operations on non-atomic cells obey the usual-looking Hoare triples, but an important departure from sequential consistency is that their associated points-to assertion $\ell[k] \rightsquigarrow_{\mathrm{na}} v$ is subjective: the knowledge of the latest value v of a non-atomic cell $\ell[k]$ is relative to the ambient view.[8]

On the other hand, the points-to assertion $\ell[k] \rightsquigarrow_{\mathrm{at}} \langle v, \mathcal{V} \rangle$ that represents an atomic cell is objective: this implies that an atomic cell $\ell[k]$ stores a single value v on which all threads agree. In addition, an atomic cell also stores a view $\mathcal{V}$. As can be seen in the triples of atomic operations, this view accumulates the knowledge of all writers and transmits this knowledge to readers. This means that there is a happens-before relationship from a write operation to a read operation which reads from it (in a release-acquire fashion). In fact, atomic accesses are the prime means of inter-thread synchronization in Multicore OCaml.

The last two triples say that a failed CAS operation behaves as an atomic read (returning `false`), while a successful CAS operation behaves as the combination of an atomic read and an atomic write (returning `true`).

3.2 Overview of the Data Structure

The code of the concurrent queue library we consider appears in Figure 8. It uses a ring buffer of fixed capacity $C \geq 1$. The buffer is represented by two arrays of length C, `statuses` and `items`; in each slot (whose offset ranges from 0 included to C excluded), the buffer thus stores a *status* in an atomic field and an *item* in a non-atomic field. The data structure also has two integers stored in atomic references, `head` and `tail`.

Items are identified by their *rank* (starting at zero) of insertion in the queue since its creation. The item of rank k is stored in slot "$k \bmod C$", which from now on we denote as $\widehat{k}$. The reference `head` stores the number of items that have been enqueued since the creation of the queue, including those that have since been dequeued. In other words, it is the rank of the next item to be enqueued. Similarly, `tail` stores the number of items that have been dequeued since the creation of the queue. In other words, it is the rank of the next item to be dequeued.

Each slot is in one of two states: either it is *occupied*, meaning that it stores some item of the queue; or it is *available*, meaning that the value it stores is irrelevant. In addition, for the concurrent queue operations to work properly, we must remember for which rank each slot was last used.[9] The status encodes this information in a single integer, as follows:

- an even status $2k$ indicates that slot $\widehat{k}$ is available for storing a future item of rank k;
- an odd status $2k + 1$ indicates that slot $\widehat{k}$ is currently occupied by the item of rank k.

Let h and t be the value of references `head` and `tail`, respectively. At any time, the ranks of items that are stored in the queue—or in the process of being stored—range from t included to h excluded, and there cannot be more than C such items. Thus, an invariant property of the queue is:

$$0 \leq t \leq h \leq t + C$$

3.3 Explanation of the Code

The function `enqueue` repeatedly calls `try_enqueue` until it succeeds; the latter can fail either because the buffer is full or because of a competing enqueuer.[10]

When calling `try_enqueue`, we start by reading the current value h of the reference `head`. To check that slot $\widehat{h}$ is available for rank h, we read its status. If it differs from $2h$, we fail: a status

[8]This single-value points-to assertion effectively forbids reasoning about races on non-atomic cells. The operational semantics of Dolan et al. [2018] is more general, as it gives a well-defined (albeit nondeterministic) semantics to racy uses of non-atomic cells.

[9]Actually, we need not remember the full rank k: only the cycle, $k \div C$, is needed.

[10]The code does not distinguish between these two causes, but this is feasible with only one more test.

```
let make () =
    let tail = $\mathtt{ref}_{\mathrm{at}}$ 0 in
    let head = $\mathtt{ref}_{\mathrm{at}}$ 0 in
    let items = $\mathtt{array}_{\mathrm{na}}[C]$ () in
    let statuses = $\mathtt{array}_{\mathrm{at}}[C]$ 0 in
    for $i$ from 0 to $C - 1$ do
        statuses$[i]_{\mathrm{na}} \leftarrow 2i$
    done;
    (tail, head, statuses, items)

let try_enqueue $q$ $v$ =
    let (tail, head,
         statuses, items) = $q$ in
    let $h$ = $!_{\mathrm{at}}$ head in
    let $s$ = statuses$[h \bmod C]_{\mathrm{at}}$ in
    if $s = 2h \land$ CAS head $h$ $(h + 1)$ then

        items$[h \bmod C]_{\mathrm{na}} \leftarrow v$;
        statuses$[h \bmod C]_{\mathrm{at}} \leftarrow 2h + 1$;
        true
    else
        false

let try_dequeue $q$ =
    let (tail, head,
         statuses, items) = $q$ in
    let $t$ = $!_{\mathrm{at}}$ tail in
    let $s$ = statuses$[t \bmod C]_{\mathrm{at}}$ in
    if $s = 2t + 1 \land$ CAS tail $t$ $(t + 1)$ then
        let $v$ = items$[t \bmod C]_{\mathrm{na}}$ in
        items$[t \bmod C]_{\mathrm{na}} \leftarrow$ ();
        statuses$[t \bmod C]_{\mathrm{at}} \leftarrow 2(t + C)$;
        Some $v$
    else
        None

let rec enqueue $q$ $v$ =
    if try_enqueue $q$ $v$
      then  ()
      else  enqueue $q$ $v$

let rec dequeue $q$ =
    match try_dequeue $q$ with
     | Some $v$ $\longrightarrow v$
     | None    $\longrightarrow$ dequeue $q$
```

Fig. 8. Implementation of the bounded queue

less than $2h$ indicates that the buffer is full[11], a status greater than $2h$ indicates interference from another competing enqueuing thread, which has been attributed rank h before we have.

If the status is $2h$, then we try to increment head from h to $h + 1$. If the CAS fails, we fail: again, another competing enqueuer has been attributed rank h.

If the CAS succeeds, then we are attributed rank h and can proceed to inserting an item into slot $\widehat{h}$. As we will explain later, this implies that its status has not changed since we read it: the slot is still available. We write the new item, and then we update the status accordingly. This update must come last, as it serves as a signal that the slot is now occupied with an item and ready for dequeuers.

[11] Technically, the public state of the queue may contain less than C elements in this case, so that we may consider it is not full. Here, by "full" we mean that the next buffer slot is either not reclaimed by anyone or still in the process of being emptied by another thread. Even though the buffer is "full", it may have available slots if dequeuing has completed more rapidly in these other slots.

Similarly, dequeue repeatedly calls `try_dequeue` until it succeeds; the latter works analogously to `try_enqueue`,[12] and can fail either because the buffer is empty or because of a competing dequeuer.[10]

3.4 Monotonicity of the Internal State of the Queue

Once the queue has been created, the reference head is only accessed from function `try_enqueue`. The only place where it is modified is the compare-and-set operation in this function, which attempts to increment it by one, using a compare-and-set operation. Hence this counter is strictly monotonic, and we can regard a successful increment from h to $h + 1$ as uniquely attributing rank h to the candidate enqueuer. Only then it is allowed to write into slot $\widehat{h}$.

Similarly, `tail` is only accessed from function `try_dequeue`, is strictly monotonic, and a successful increment from t to $t + 1$ uniquely attributes rank t to the candidate dequeuer. Only then it is allowed to write into slot $\widehat{t}$.

The status of a given slot is strictly monotonic too. Indeed, there are two places where it is updated. As an enqueuer, when we write $2h + 1$, no other enqueuer updated the status since we read it to be $2h$, because only we have been attributed rank h. In particular, it remained even, so no dequeuer tried to obtain rank h and update the status of slot $\widehat{h}$. Hence, the status is still $2h$ when we overwrite it with $2h + 1$. Symmetrically, the status is still $2t + 1$ when a dequeuer overwrites it with $2(t + C)$.

3.5 Notes on Contention in the Queue

A noteworthy feature of this implementation is that it tries to limit competition between enqueuers and dequeuers. Indeed, enqueuers and dequeuers generally operate on separate references: enqueuers never access `tail` and dequeuers never access head. Hence in favorable situations—when the buffer is neither empty nor full—there are no enqueuer-dequeuer competitions beyond ones between an enqueuer and a dequeuer of the same rank.

A weakness of this implementation, however, is that it does not enjoy any non-blocking property [Fraser 2004, Chapter 2]: if an enqueuer or a dequeuer halts after it has been attributed a rank but before it updates the corresponding slot, then after some time, any other thread trying to enqueue or dequeue fails. »

4 PROOF OF THE SPECIFICATION FOR THE RING BUFFER

We now turn to proving the following.

Theorem 4.1. *There exist predicates* IsQueue *and* QueueInv *such that the implementation shown in Figure 8 (§3) satisfies the functional specification appearing in Figure 5 (§2.3).*

In the Iris methodology, which Cosmo is based on, concurrent protocols are established thanks to *ghost state* and *invariants*. Ghost state in Iris is a flexible tool for defining custom resources. It takes values from algebraic structures called CMRAs; for the purpose of this paper, it is enough to think of a CMRA as a set equipped with a binary composition operation $(\cdot)$ that is partial, associative and commutative. Ghost state values are assigned to ghost variables. The separation logic assertion $\boxed{a}^{\gamma}$, where a is an element of some CMRA, intuitively means that we own a fragment of the ghost variable γ and that this fragment has value a. Unlike what happens with a traditional points-to assertion, the ownership *and value* of a ghost variable can be split into fragments according to the

[12]Overwriting the extracted value with a unit value `()` is unnecessary for functional correctness but it prevents memory leaks.

$$\begin{array}{rcl} \gamma & : & \textsc{Auth}(\textsc{Ex}(\textsc{View}{\times}\textsc{View}{\times}\textsc{List}(\textsc{Val}{\times}\textsc{View}))) \\ \mathsf{IsQueue}\ \gamma\ \mathcal{T}\ \mathcal{H}\ [(v_0,\mathcal{V}_0),\dots,(v_{n-1},\mathcal{V}_{n-1})] & \triangleq & \boxed{\circ\ (\mathcal{T},\mathcal{H},[(v_0,\mathcal{V}_0),\dots,(v_{n-1},\mathcal{V}_{n-1})])}^{\gamma} \end{array}$$

(a) Ghost state describing the public state

$$\begin{array}{rcl} \textsc{StatusWithView} & \triangleq & (\mathbb{Z}\times\textsc{View},\sqsubseteq,\sqcup) \\ (s_1,\mathcal{S}_1)\sqsubseteq(s_2,\mathcal{S}_2) & \triangleq & s_1 < s_2 \lor (s_1 = s_2 \land \mathcal{S}_1 \sqsupseteq \mathcal{S}_2) \\ (s_1,\mathcal{S}_1)\sqcup(s_2,\mathcal{S}_2) & \triangleq & \begin{cases} (s_2,\mathcal{S}_2) & \text{if } s_1 < s_2 \\ (s,\mathcal{S}_1\sqcap\mathcal{S}_2) & \text{if } s_1 = s_2 = s \\ (s_1,\mathcal{S}_1) & \text{if } s_1 > s_2 \end{cases} \end{array}$$

(b) Semi-lattice defining an order on status-view pairs

$$\begin{array}{rcl} \gamma_{\text{mono}} & : & \mathbb{Z}\xrightarrow{\text{fin}}\textsc{Auth}(\textsc{StatusWithView}) \\ \mathsf{Witness}\ \gamma_{\text{mono}}\ i\ (s,\mathcal{S}) & \triangleq & \boxed{\{i\mapsto\circ\ (s,\mathcal{S})\}}^{\gamma_{\text{mono}}} \end{array}$$

(c) Ghost state reflecting the monotonicity of statuses

$$\begin{array}{rcl} \gamma_{\text{tokens}} & : & \textsc{Auth}\left(\mathbb{Z}\xrightarrow{\text{fin}}\textsc{Ex}(\textsc{Unit}\ +\ \textsc{Val}\times\textsc{View})\right) \\ \mathsf{TokenR}\ \gamma_{\text{tokens}}\ k & \triangleq & \boxed{\circ\ \{k\mapsto()\}}^{\gamma_{\text{tokens}}} \\ \mathsf{TokenW}\ \gamma_{\text{tokens}}\ k\ (v,\mathcal{V}) & \triangleq & \boxed{\circ\ \{k\mapsto(v,\mathcal{V})\}}^{\gamma_{\text{tokens}}} \end{array}$$

(d) Ghost state implementing tokens

$$\begin{array}{rcl} \mathsf{QueueInv}\ q\ \gamma & \triangleq & \left\{\begin{array}{l} \exists\gamma_{\text{mono}},\gamma_{\text{tokens}}. \\ \quad \boxed{\mathsf{QueueInvInner}\ q\ \gamma\ \gamma_{\text{mono}}\ \gamma_{\text{tokens}}} \end{array}\right. \\ \mathsf{QueueInvInner}\ q\ \gamma\ \gamma_{\text{mono}}\ \gamma_{\text{tokens}} & \triangleq & \end{array}$$

$$\left\{\begin{array}{l} \exists \mathtt{tail},\mathtt{head},\mathtt{statuses},\mathtt{items}. \\ \exists t,\mathcal{T},h,\mathcal{H},(v_t,\mathcal{V}_t),\dots,(v_{h-1},\mathcal{V}_{h-1}),(s_0,\mathcal{S}_0),\dots,(s_{C-1},\mathcal{S}_{C-1}). \\ \ast\left\{\begin{array}{l} q = (\mathtt{tail},\mathtt{head},\mathtt{statuses},\mathtt{items}) \\ 0\le t\le h\le t+C \\ \mathtt{tail}\rightsquigarrow_{\text{at}}\langle t,\mathcal{T}\rangle\ \ast\ \mathtt{head}\rightsquigarrow_{\text{at}}\langle h,\mathcal{H}\rangle \\ \mathtt{statuses}\rightsquigarrow^{*}_{\text{at}}[(s_0,\mathcal{S}_0),\dots,(s_{C-1},\mathcal{S}_{C-1})] \\ \boxed{\bullet\ (\mathcal{T},\mathcal{H},[(v_t,\mathcal{V}_t),\dots,(v_{h-1},\mathcal{V}_{h-1})])}^{\gamma} \\ \boxed{\{i\mapsto\bullet\,(s_i,\mathcal{S}_i)\mid 0\le i<C\}}^{\gamma_{\text{mono}}} \\ \boxed{\bullet\left(\begin{array}{rlcr} & \{k\mapsto()\} & \mid & h-C\le k<t\} \\ \uplus & \{k\mapsto(v_k,\mathcal{V}_k) & \mid & t\le k<h\} \end{array}\right)}^{\gamma_{\text{tokens}}} \\ \mathop{\Huge\ast}_{h-C\le k<t}\ \mathsf{Available}\ k\ (s_{\widehat{k}},\mathcal{S}_{\widehat{k}}) \qquad\quad \lor\ \ s_{\widehat{k}} = 2k+1 \\ \mathop{\Huge\ast}_{t\le k<h}\ \mathsf{Occupied}\ k\ (s_{\widehat{k}},\mathcal{S}_{\widehat{k}})\ (v_k,\mathcal{V}_k)\ \ \lor\ \ s_{\widehat{k}} = 2k \end{array}\right. \end{array}\right.$$

$$\begin{array}{rcl} \mathsf{Available}\ k\ (s,\mathcal{S}) & \triangleq & \ast\left\{\begin{array}{l} s = 2(k+C)\ \ast\ (\mathtt{items}[\widehat{k}]\rightsquigarrow_{\text{na}} -)\ @\ \mathcal{S} \\ \mathsf{TokenR}\ \gamma_{\text{tokens}}\ k \end{array}\right. \\ \mathsf{Occupied}\ k\ (s,\mathcal{S})\ (v,\mathcal{V}) & \triangleq & \ast\left\{\begin{array}{l} s = 2k+1\qquad\ast\ (\mathtt{items}[\widehat{k}]\rightsquigarrow_{\text{na}} v)\ @\ \mathcal{S} \\ \mathsf{TokenW}\ \gamma_{\text{tokens}}\ k\ (v,\mathcal{V})\ \ast\ \mathcal{V}\sqsubseteq\mathcal{S} \end{array}\right. \end{array}$$

(e) Internal invariant of the queue

Fig. 9. Definitions of assertions intervening in the proof of the bounded queue

composition operation of the CMRA: $\boxed{a \cdot b}^{\gamma} \dashv\vdash \boxed{a}^{\gamma} * \boxed{b}^{\gamma}$. In Cosmo, ghost state is objective, as it is independent from the physical state of the memory.

Ghost state can be coupled with invariants such as the ones presented in §2.2 to describe protocols that threads must follow to access shared resources. In Figure 9, one can see how this methodology is used for describing the internal protocol of the queue: the persistent predicate QueueInv $q\ \gamma$ is in fact an invariant; the exclusive representation predicate IsQueue $\gamma\ \mathcal{T}\ \mathcal{H}\ [(v_0, \mathcal{V}_0), \ldots, (v_{n-1}, \mathcal{V}_{n-1})]$ is defined using ghost state, as are several internal resources. We detail these definitions in the following sections.

4.1 Public State

The assertion IsQueue $\gamma\ \mathcal{T}\ \mathcal{H}\ [(v_0, \mathcal{V}_0), \ldots, (v_{n-1}, \mathcal{V}_{n-1})]$, defined in Figure 9a, exposes to the user the public state of the queue. This public state, as motivated in §2.3, is composed of the tail view, the head view, and the list of current items with their views. It is tied to the internal state of the queue via the use of an authoritative ghost state, stored in a ghost variable γ. More precisely, the public state is kept in sync with the values which appear in an authoritative assertion $\boxed{\bullet\ (\mathcal{T}, \mathcal{H}, [(v_0, \mathcal{V}_0), \ldots, (v_{n-1}, \mathcal{V}_{n-1})])}^{\gamma}$, the latter being owned by the internal invariant.

The two assertions satisfy the properties shown in Figure 10a. Rule IsQueue-Agree asserts that the state known to the invariant (first premise) is identical to that known to the representation predicate (second premise). Rule IsQueue-Update asserts that, whenever we own both the representation predicate and its authoritative counterpart, we can update the public state to any other value by taking a *ghost update* step. Such a ghost update is allowed by the Iris modality denoted $\dot{\Rrightarrow}$.

We achieve these properties by using an adequate CMRA for the values of the ghost variable γ. This CMRA is built by composing several classical Iris constructs: the exclusive CMRA $\mathrm{Ex}(S)$, and the authoritative CMRA $\textsc{Auth}(M)$. We do not explain the construction in more detail; we refer the interested reader to the documentation of Iris [Jung et al. 2018].

It is worth remarking that this construction makes the representation predicate exclusive: it is absurd to own simultaneously two assertions of the form IsQueue γ − − −.

4.2 Internal Invariant

Along with the exclusive representation predicate IsQueue $\gamma\ \mathcal{T}\ \mathcal{H}\ [(v_0, \mathcal{V}_0), \ldots, (v_{n-1}, \mathcal{V}_{n-1})]$, we provide the user with a persistent assertion QueueInv $q\ \gamma$ defined in Figure 9e. It contains the internal invariant governing the queue q, whose public state is exposed via the ghost variable γ. In addition to the public state, there are two more ghost variables, named γ_{mono} and γ_{tokens}, which are hidden to the user of the queue but needed internally. Thus they are existentially quantified in this persistent assertion. We will explain the purpose and meaning of these ghost variables in a moment. For now, we look at the internal invariant, QueueInvInner $q\ \gamma\ \gamma_{\text{mono}}\ \gamma_{\text{tokens}}$.

This invariant owns most of the physical locations of the queue: `tail`, `head`, `statuses`, and some parts of the array `items`. Recall that points-to assertions for atomic cells are objective and can be placed inside an invariant. The array-points-to assertion `statuses` $\rightsquigarrow^{*}_{\text{at}} [(s_0, \mathcal{S}_0), \ldots, (s_{C-1}, \mathcal{S}_{C-1})]$ is a shorthand for the following iterated conjunction:

$$\mathop{\Large *}_{0 \le i < C} \texttt{statuses}[i] \rightsquigarrow_{\text{at}} \langle s_i, \mathcal{S}_i \rangle$$

Also, since we encode references as arrays of length one, we write `tail` $\rightsquigarrow_{\text{at}} \langle t, \mathcal{T} \rangle$ as a shorthand for `tail[0]` $\rightsquigarrow_{\text{at}} \langle t, \mathcal{T} \rangle$.

Apart from this physical state, the invariant also stores ghost state. It owns the authority on all three ghost variables, γ, γ_{mono} and γ_{tokens}. The authority of γ is simple: it ties internal values to the public state of the queue, as explained earlier. We now explain the other two pieces of ghost state.

IsQueue-Agree

$$\frac{\boxed{\bullet\ (\mathcal{T},\mathcal{H},[(v_0,\mathcal{V}_0),\ldots,(v_{n-1},\mathcal{V}_{n-1})])}^{\gamma} \ * \ \mathsf{IsQueue}\ \gamma\ \mathcal{T}'\ \mathcal{H}'\ \left[(v'_0,\mathcal{V}'_0),\ldots,(v'_{n-1},\mathcal{V}'_{n-1})\right]}{\mathcal{T}=\mathcal{T}' \ \land\ \mathcal{H}=\mathcal{H}' \ \land \forall i.\ \ v=v'_i \ \land\ \mathcal{V}_i=\mathcal{V}'_i}$$

IsQueue-Update

$$\frac{\boxed{\bullet\ (\mathcal{T},\mathcal{H},[(v_0,\mathcal{V}_0),\ldots,(v_{n-1},\mathcal{V}_{n-1})])}^{\gamma} \ * \ \mathsf{IsQueue}\ \gamma\ \mathcal{T}\ \mathcal{H}\ [(v_0,\mathcal{V}_0),\ldots,(v_{n-1},\mathcal{V}_{n-1})]}{\dot{\Rrightarrow}\boxed{\bullet\ \left(\mathcal{T}',\mathcal{H}',\left[(v'_0,\mathcal{V}'_0),\ldots,(v'_{n-1},\mathcal{V}'_{n-1})\right]\right)}^{\gamma} \ * \ \mathsf{IsQueue}\ \gamma\ \mathcal{T}'\ \mathcal{H}'\ \left[(v'_0,\mathcal{V}'_0),\ldots,(v'_{n-1},\mathcal{V}'_{n-1})\right]}$$

(a) Properties of the representation predicate

Witness-Persistent

$$\frac{}{\mathsf{persistent}(\mathsf{Witness}\ \gamma_{\text{mono}}\ i\ (s,\mathcal{S}))}$$

Witness-Order

$$\frac{\boxed{\{i \mapsto \bullet\ (s,\mathcal{S})\}}^{\gamma_{\text{mono}}} \ * \ \mathsf{Witness}\ \gamma_{\text{mono}}\ i\ (s',\mathcal{S}')}{(s',\mathcal{S}') \sqsubseteq (s,\mathcal{S})}$$

Witness-Update

$$\frac{\boxed{\{i \mapsto \bullet\ (s,\mathcal{S})\}}^{\gamma_{\text{mono}}} \ * \ (s,\mathcal{S}) \sqsubseteq (s',\mathcal{S}')}{\dot{\Rrightarrow}\boxed{\{i \mapsto \bullet\ (s',\mathcal{S}')\}}^{\gamma_{\text{mono}}} \ * \ \mathsf{Witness}\ \gamma_{\text{mono}}\ i\ (s',\mathcal{S}')}$$

(b) Properties of witnesses

Token-Exclusive-RR

$$\frac{\mathsf{TokenR}\ \gamma_{\text{tokens}}\ k \ * \ \mathsf{TokenR}\ \gamma_{\text{tokens}}\ k'}{\widehat{k} \neq \widehat{k'}}$$

Token-Exclusive-RW

$$\frac{\mathsf{TokenR}\ \gamma_{\text{tokens}}\ k \ * \ \mathsf{TokenW}\ \gamma_{\text{tokens}}\ k'\ (v',\mathcal{V}')}{\widehat{k} \neq \widehat{k'}}$$

Token-Exclusive-WW

$$\frac{\mathsf{TokenW}\ \gamma_{\text{tokens}}\ k\ (v,\mathcal{V}) \ * \ \mathsf{TokenW}\ \gamma_{\text{tokens}}\ k'\ (v',\mathcal{V}')}{\widehat{k} \neq \widehat{k'}}$$

TokenR-Agree

$$\frac{\boxed{\bullet\, m}^{\gamma_{\text{tokens}}} \ * \ \mathsf{TokenR}\ \gamma_{\text{tokens}}\ k}{m(k) = ()}$$

TokenW-Agree

$$\frac{\boxed{\bullet\, m}^{\gamma_{\text{tokens}}} \ * \ \mathsf{TokenW}\ \gamma_{\text{tokens}}\ k\ (v,\mathcal{V})}{m(k) = (v,\mathcal{V})}$$

Token-Update-RW

$$\frac{\boxed{\bullet\, m}^{\gamma_{\text{tokens}}} \ * \ \mathsf{TokenR}\ \gamma_{\text{tokens}}\ (k-C)}{\dot{\Rrightarrow}\boxed{\bullet\, m[k-C \mapsto \bot][k \mapsto (v,\mathcal{V})]}^{\gamma_{\text{tokens}}} \ * \ \mathsf{TokenW}\ \gamma_{\text{tokens}}\ k\ (v,\mathcal{V})}$$

Token-Update-WR

$$\frac{\boxed{\bullet\, m}^{\gamma_{\text{tokens}}} \ * \ \mathsf{TokenW}\ \gamma_{\text{tokens}}\ k\ (v,\mathcal{V})}{\dot{\Rrightarrow}\boxed{\bullet\, m[k \mapsto ()]}^{\gamma_{\text{tokens}}} \ * \ \mathsf{TokenR}\ \gamma_{\text{tokens}}\ k}$$

(c) Properties of tokens

Fig. 10. Axiomatic description of the ghost state of the queue

4.3 Monotonicity of Statuses

The purpose of the ghost variable γ_{mono} is to reflect the fact that statuses are monotonic. More precisely, they are *strictly* monotonic: every write to a status cell necessarily increases its value. As a consequence, as long as the value of a status cell has not increased, we know that no write happened to it and, in particular, that the view that it stores has not increased either. In other words, we have the monotonicity of the value-view pair stored in a status cell, for the lexicographic order where the order on views is reversed:

$$(s_1, \mathcal{S}_1) \sqsubseteq (s_2, \mathcal{S}_2) \iff s_1 < s_2 \lor (s_1 = s_2 \land \mathcal{S}_1 \sqsupseteq \mathcal{S}_2)$$

This stronger monotonicity property will be used in proofs, and specifying it is thus an additional requirement of working with a weak memory model.

To reflect monotonicity of the status of offset i, we use two assertions, $\boxed{\{i \mapsto \bullet\ (s, \mathcal{S})\}}^{\gamma_{\text{mono}}}$ and Witness $\gamma_{\text{mono}}\ i\ (s, \mathcal{S})$, connected via a ghost variable γ_{mono}. Relevant definitions appear in Figure 9c. The first assertion, owned by the invariant of the queue is connected by the invariant to the value-view pair stored in the status cell. It is exclusive: for any offset i, two assertions of the form $\boxed{\{i \mapsto \bullet -\}}^{\gamma_{\text{mono}}}$ cannot hold simultaneously. The second assertion Witness $\gamma_{\text{mono}}\ i\ (s, \mathcal{S})$ means that the value-view pair stored in the status cell is at least $(s, \mathcal{S})$. Importantly, a witness assertion is persistent: once it has been established, it remains true forever and can be duplicated at will.

We thus have the properties summarized in Figure 10b. Rule Witness-Persistent is the persistence just mentioned. Rule Witness-Order asserts that a witness gives a lower bound on what the status cell currently stores. Rule Witness-Update asserts that we can update a status cell to any larger (or equal) content, and obtain a witness for that content.

We achieve these properties by constructing an adequate CMRA for the values taken by the ghost variable γ_{mono}. Again, we will not explain standard Iris constructs here, except for one point. The construction involves building a CMRA whose carrier set is $\mathbb{Z} \times \text{View}$, the set of status-view pairs, and whose inclusion order[13] coincides with the desired order $\sqsubseteq$. A general recipe for deriving a CMRA structure with a given inclusion order, if that order admits binary joins, consists in taking the join operation as the composition of the CMRA [Timany and Birkedal 2021]. In this case, we equip the product set $\mathbb{Z} \times \text{View}$ with the join-semilattice structure whose definition appears in Figure 9b.

4.4 Available and Occupied Slots

In Figure 9e, the last two lines of the invariant describe the state of each slot. For clarity, we introduce two abbreviations: the assertion Available $k\ (s, \mathcal{S})$ represents slot $\widehat{k}$ being available for a future item of rank $k + C$; the assertion Occupied $k\ (s, \mathcal{S})\ (v, \mathcal{V})$ represents slot $\widehat{k}$ being occupied by the item of rank k, whose value is v with the associated view $\mathcal{V}$. In these two abbreviations, the status field of the slot has value s and stores view $\mathcal{S}$. These abbreviations are also where we keep the ownership of the *non-atomic* cell `items`$[\widehat{k}]$, via a points-to assertion.

Recall that, in Cosmo, unlike an atomic points-to assertion, a non-atomic points-to assertion is *subjective*: its truth depends on the view of the subject thread. As a consequence, it cannot be placed in an invariant as is. In order to share this assertion, we must explicitly indicate at which view it holds. This is the purpose of the @ connective.

At which view can we own a non-atomic memory cell? At a view which contains the latest write event to that cell. Fortunately, in our case, any thread—enqueuer or dequeuer—which writes to the non-atomic cell `items`$[\widehat{k}]$ then writes to the atomic cell `statuses`$[\widehat{k}]$. Thus it adds its knowledge, including its own write to the item field, to the view $\mathcal{S}$ stored by the status field.

[13] For two elements a and b of a CMRA M, we say that b is included in a if there exists some c such that $a = b \cdot c$.

With all this said, a first attempt at representing the buffer might look as follows:

$$\begin{aligned}
\mathsf{QueueInvInner}\ q\ \gamma\ \gamma_{\text{mono}}\ \gamma_{\text{tokens}} &\overset{?}{=} \ast\begin{cases} \vdots \\ \displaystyle\mathop{\ast}_{h-C\le k<t} \mathsf{Available}\ k\ (s_{\widehat{k}}, \mathcal{S}_{\widehat{k}}) \\ \displaystyle\mathop{\ast}_{t\le k<h} \mathsf{Occupied}\ k\ (s_{\widehat{k}}, \mathcal{S}_{\widehat{k}})\ (v_k, \mathcal{V}_k) \end{cases} \\
\mathsf{Available}\ k\ (s, \mathcal{S}) &\overset{?}{=} s = 2(k+C) \ast (\texttt{items}[\widehat{k}] \rightsquigarrow_{\text{na}} -) \mathbin{@} \mathcal{S} \\
\mathsf{Occupied}\ k\ (s, \mathcal{S})\ (v, \mathcal{V}) &\overset{?}{=} s = 2k+1 \quad \ast (\texttt{items}[\widehat{k}] \rightsquigarrow_{\text{na}} v) \mathbin{@} \mathcal{S} \ast \mathcal{V} \sqsubseteq \mathcal{S}
\end{aligned}$$

That is, we describe the C slots by ranging from $h-C$ to h. The indices from $h-C$ to t correspond to available slots, while indices from t to h correspond to slots occupied by the items of the queue. In both cases, we own the item field at the view $\mathcal{S}$ which is stored in the corresponding status field. The item field of an available slot stores an arbitrary value, while for an occupied slot it stores the item v.

An occupied slot should also carry the view $\mathcal{V}$ which the queue is supposed to transfer from the enqueuer to the dequeuer alongside item v. This again relies on the view $\mathcal{S}$: the enqueuer adds $\mathcal{V}$ to $\mathcal{S}$ when updating the status, and the dequeuer adds $\mathcal{S}$ into its own view when reading the status; so, to retrieve $\mathcal{V}$, it is enough to state the inclusion $\mathcal{V} \sqsubseteq \mathcal{S}$.

The tentative invariant stated above, however, is not correct: while an invariant has to hold at any point of the execution, the assertion above is temporarily invalidated when a thread enqueues or dequeues. Specifically, the thread breaks the assertion when it increments `head` or `tail`, thus committing to enqueuing or dequeuing, until it updates the status of the corresponding slot. It is thus necessary to represent slots which are in a temporary state. In the actual invariant shown in Figure 9e, slots from $h-C$ to t are either available or in a temporary state where they appear as occupied ($s_{\widehat{k}} = 2k+1$), until a dequeuer finishes emptying them; slots from t to h are either occupied or in a temporary state where they appear as available ($s_{\widehat{k}} = 2k$), until an enqueuer finishes filling them.

When an enqueuer or dequeuer moves a slot into a temporary state, it takes ownership of its item field, so that it can write to it. Hence the invariant does not have the corresponding points-to assertion. The thread must give it back when updating the status.

4.5 Slot Tokens

This time frame—when a slot is in a temporary state—is also when the last piece of ghost state, stored in the ghost variable γ_{tokens}, intervenes. Other threads can make the queue progress between the moment when an enqueuer is attributed rank k, and the moment when it returns the updated slot to the invariant. An enqueuer needs the assurance that the queue has not gotten too far and attributed the slot on which it was working to a dequeuer, or to another enqueuer in a subsequent cycle.

To this effect, we start by stating how advances of the `head` and `tail` are limited with respect to one another; indeed, we prove these inequalities as part of the invariant:

$$0 \le t \le h \le t + C$$

We also maintain in existence one token for each rank from $h-C$ to h. These tokens are exclusive assertions, and there cannot exist two tokens whose ranks are congruent modulo C. Hence the token of rank k is enough to grant unique write access to slot $\widehat{k}$. We use it as follows.

(1) When an enqueuer is attributed rank k, it borrows a newly created token of the same rank.

(2) When returning the updated slot to the invariant, the enqueuer also returns the token; from that moment the token is thus kept in the assertion $\mathsf{Occupied}\ k\ (s, \mathcal{S})\ (v, \mathcal{V})$.
(3) When a dequeuer is attributed rank k, it claims that assertion and borrows the token.
(4) When returning the updated slot to the invariant, the dequeuer also returns the token; from that moment the token is thus kept in the assertion $\mathsf{Available}\ k\ (s, \mathcal{S})$.

In step 1, the token is created while destructing the token of rank $k - C$ taken from the assertion $\mathsf{Available}\ (k - C)\ -$, which represents the available cell that the thread claims for enqueuing.

To be able to distinguish between the two temporary states (enqueuing and dequeuing), we give the token a flavor: from steps 1 to 2 it is a *write token*; from steps 3 to 4 it is a *read token*. At any moment, there are read tokens from rank $h - C$ to t, and write tokens from t to h.

We have a last requirement: when an enqueuer is attributed rank k, the new item is added to the public state immediately—the CAS operation on head is the commit point of enqueuing—even though the enqueuer has not actually written the item yet. When it finally returns the updated slot, the enqueuer has lost track of the public state, which may have continued to progress in the meantime. At that moment, it thus needs a reminder that the item it just wrote is indeed the one it was expected to write. We implement this by adding the value v—and view $\mathcal{V}$—of the item as a payload to the write token.

The read token of rank k is denoted by $\mathsf{TokenR}\ \gamma_{\text{tokens}}\ k$, while the write token of rank k with payload v and $\mathcal{V}$ is denoted by $\mathsf{TokenW}\ \gamma_{\text{tokens}}\ k\ (v, \mathcal{V})$. Their authoritative counterpart, owned by the invariant, is an assertion of the form $\boxed{\bullet\, m}^{\gamma_{\text{tokens}}}$ where m is a finite map. Its domain is the range $[h - C, h)$ of ranks which have a token, and its images are the payload (considering that read tokens bear a payload of $()$). In the invariant, the value of the map is connected to that of the public state.

The assertions are defined in Figure 9d and satisfy the properties in Figure 10c. The first three properties say that tokens are exclusive. The next two say that tokens agree with the authoritative counterpart, hence with the public state. Rule Token-Update-RW corresponds to step 1 in the list above, where we create a write token of rank k by destructing a read token of rank $k - C$. Likewise, rule Token-Update-WR corresponds to step 3, where we turn a write token into a read token of the same rank.

In addition to these rules, the finite map described in the internal invariant is such that, whenever we own a read token (respectively a write token), the rank of this token necessarily lies in the range $[h - C, t)$ (respectively $[t, h)$), where t and h are the values of `tail` and head which are existentially quantified in the invariant. Thanks to that property, at step 4 (respectively 2), when a dequeuer (respectively an enqueuer) returns the token, it knows that the rank it has been operating on is still in the right range—in other words, that the queue has not advanced too far while the thread was working.

There are more properties that are invariants of the queue, and thus could be stated and verified. However, they are not needed to prove that the code satisfies its specification. For example, the fact that `tail` and head are strictly monotonic, and the fact that statuses are non-negative, are not explicitly used.

4.6 Logical Atomicity

The specification that we wish to prove is a logically atomic Hoare triple. The definition of such triples for Iris is given by Jung et al. [2015, §7] and further refined by Jung [2019]. It turns out that this definition can be ported as is using the connectives of Cosmo. As we will see in §4.7 and §5.3, the logically atomic triples so defined can be proved and are sufficient for interesting clients. We do not attempt to replicate in this paper the full definition. An approximate definition that suffices to capture the essence of logical atomicity, and to understand our proof, is:

$$\langle x.\, P\rangle\; e\; \langle \Phi\rangle \quad \triangleq \quad \forall \Psi,\, \left[{}^{\top}{\Rrightarrow}^{\emptyset}\, \exists x.\; P \;*\; \left(\forall v.\, \Phi\, v \;-\!\!*\; {}^{\emptyset}{\Rrightarrow}^{\top}\, \Psi\, v\right)\right] \;-\!\!*\; \mathsf{wp}\; e\; \{\Psi\}$$

In this formula, the variable P is a Cosmo assertion (of type vProp); the variables Φ and Ψ are predicates on values (of type Val $\rightarrow$ vProp); P and Φ may refer to the name x. The assertion $\mathsf{wp}\; e\; \{\Psi\}$ is the weakest precondition for program e and postcondition Ψ (in Iris, Hoare triples are syntactic sugar for weakest preconditions).

The purpose of a logically atomic triple is to give a specification to a non-atomic program e *as if it were atomic*. In practice, we require that the proof of e accesses the precondition and turns it into the postcondition in *one atomic step* only, which we call the commit point of this logically atomic program. That is, if e satisfies the triple $\langle x.\, P\rangle\; e\; \langle \Phi\rangle$, then it can perform several steps of computation but, as soon as it accesses the resource P, it must return the resource Φ in the same step of computation.[14] Once it has done so, e can perform further computation steps but P is not available anymore. As explained in §2.2, thanks to this constraint, the client of this specification can open invariants around e as if e were atomic.

To capture this atomicity requirement, we ask the proof of the logically atomic triple for e to be valid for any postcondition Ψ chosen by the client. Given that Ψ is arbitrary, the only means of establishing this postcondition is to use the premise ${}^{\top}{\Rrightarrow}^{\emptyset}\, \exists x.\, P \;*\; (\forall v.\, \Phi\, v \;-\!\!*\; {}^{\emptyset}{\Rrightarrow}^{\top}\, \Psi\, v)$, which is known as an *atomic update*. When desired, this atomic update gives access to the precondition P for some value of x, and, in exchange for the postcondition Φ of the logically atomic triple, it returns the resource Ψ, which can then be used to finish the proof. Crucially, the *masks* $\emptyset$ and $\top$ annotating the *fancy updates* ${}^{\top}{\Rrightarrow}^{\emptyset}$ and ${}^{\emptyset}{\Rrightarrow}^{\top}$ require that the atomic update be used during one atomic step only, as required.

Using the invariant rules of Iris [Jung et al. 2018], it is easy to show that atomic updates can be used to open and close invariants. Rule LAInv follows as a corollary, rule LAHoare is immediate.

4.7 Proof of `try_enqueue`

We now outline the proof that `try_enqueue` satisfies its specification from Figure 5. The proof for `try_dequeue` is similar; those for `enqueue` and `dequeue` are deduced from the previous two by an obvious induction; and the proof of `make` is simply a matter of initializing the ghost state. The interested reader may find these proofs, conducted in the Coq proof assistant, in our repository [Mével et al. 2021].

Recalling here the specification in Figure 5, and unfolding the definition of QueueInv $q\;\gamma$, we ought to prove the following assertion:

$$\frac{\boxed{\mathsf{QueueInvInner}\; q\; \gamma\; \gamma_{\mathrm{mono}}\; \gamma_{\mathrm{tokens}}}}{\begin{array}{l} \left\langle \begin{array}{l} \mathcal{T}, \mathcal{H}, n, (v_0, \mathcal{V}_0), \ldots, (v_{n-1}, \mathcal{V}_{n-1}). \\ \qquad \mathsf{IsQueue}\; \gamma\; \mathcal{T}\;\; \mathcal{H} \qquad [(v_0, \mathcal{V}_0), \ldots, (v_{n-1}, \mathcal{V}_{n-1})] \qquad * \;\uparrow\mathcal{V} \end{array} \right\rangle \\ \quad \texttt{try\_enqueue}\; q\; v \\ \left\langle \lambda b.\, \vee \left[\begin{array}{llll} \mathsf{IsQueue}\; \gamma\; \mathcal{T}\;\; \mathcal{H} & [(v_0, \mathcal{V}_0), \ldots, (v_{n-1}, \mathcal{V}_{n-1})] & & * \; b = \texttt{false} \\ \mathsf{IsQueue}\; \gamma\; \mathcal{T}\; (\mathcal{H} \sqcup \mathcal{V}) & [(v_0, \mathcal{V}_0), \ldots, (v_{n-1}, \mathcal{V}_{n-1}), (v, \mathcal{V})] & * \;\uparrow\mathcal{H} & * \; b = \texttt{true} \end{array} \right. \right\rangle \end{array}}$$

[14] The full definition of logically atomic triples allows to access the precondition atomically before the commit point, hence without turning it into the postcondition. This is called *aborting*; it is not needed in our proof, and out of the scope of this paper.

After unfolding the logically atomic triple, we must prove wp (`try_enqueue` q v) $\{\Psi\}$ for any Ψ, when in the proof context we have the internal invariant of the queue (with ghost variables $\gamma, \gamma_{\text{mono}}, \gamma_{\text{tokens}}$) as well as the atomic update whose precondition and postcondition are that of the triple above. We then step through the program using usual weakest-precondition calculus.

The first interesting step is the atomic read of `head`. The ownership of that reference is shared in the invariant of the queue. Hence, to access it, we must open the invariant; then we get the points-to assertion, we can step through the read operation, return the points-to assertion and close the invariant again. After we have done so, and thus forgotten all the quantities which are existentially quantified inside that invariant, we learn little about the value that has just been read, excepted that it is a non-negative integer, say k.

The second interesting step is the atomic read at index $\widehat{k}$ of the array `statuses`. Again the invariant owns this cell, so we open it around the read instruction. This read yields some value s^1 and, since it is atomic, it also augments our current (thread) view with the view $\mathcal{S}^1$ which, at this moment, is stored in this cell. In other words, we gain the (persistent) assertion $\uparrow \mathcal{S}^1$. We can remember more information before closing the invariant: indeed, from the authority of γ_{mono} found in the invariant, we derive a witness for the strict monotonicity of the status that we just read: $\mathsf{Witness}\ \gamma_{\text{mono}}\ \widehat{k}\ (s^1, \mathcal{S}^1)$.

Next, the program tests whether $s^1 = 2k$. If the test fails, then the program returns `false`. In this case, we have to provide as postcondition of the logically atomic triple the untouched representation predicate that is in its precondition ($\mathsf{IsQueue}\ \gamma\ \mathcal{T}\ \mathcal{H}\ [(v_0, \mathcal{V}_0), \ldots, (v_{n-1}, \mathcal{V}_{n-1})]$). We do this by committing the atomic update in a trivial way, then conclude the proof.

If $s^1 = 2k$, the program proceeds to performing `CAS head` k $(k+1)$. To access `head`, we open the invariant again. If that operation fails, the program also returns `false` and, after closing the invariant without having updated ghost state, we conclude as before.

If the CAS succeeds, then a number of things happen logically. First, if h and t are the values of `head` and `tail` at the moment of the CAS, then $h = k$. Second, we deduce that the buffer is not full, i.e. $h < t + C$. Indeed, the invariant directly gives us $h \leq t + C$; if we had $h = t + C$, then in particular, $t \leq k - C < h$, so the invariant would own the following for slot $\widehat{k - C}$:

$$\mathsf{Occupied}\ (k-C)\ (s_{\widehat{k-C}}, \mathcal{S}_{\widehat{k-C}})\ (v_{k-C}, \mathcal{V}_{k-C})\ \vee\ s_{\widehat{k-C}} = 2(k-C)$$

Because $\widehat{k - C} = \widehat{k}$, this implies:

$$s_{\widehat{k}} = 2(k-C) + 1\ \vee\ s_{\widehat{k}} = 2(k-C)$$

In either case, we get $s_{\widehat{k}} < 2k = s^1$, which contradicts the monotonicity of the status of that slot. We derive the contradiction by combining the assertion $\mathsf{Witness}\ \gamma_{\text{mono}}\ \widehat{k}\ (s^1, \mathcal{S}^1)$ that we had since we read the status, and the authority $\boxed{\bullet\ (s_{\widehat{k}}, \mathcal{S}_{\widehat{k}})}^{\gamma_{\text{mono}}}$ that is found in the invariant.

Third, we thus know that $h - C \leq k - C < t$, so that the invariant gives us:

$$\mathsf{Available}\ (k-C)\ (s_{\widehat{k-C}}, \mathcal{S}_{\widehat{k-C}})\ \vee\ s_{\widehat{k-C}} = 2(k-C) + 1$$

Again the second disjunct is absurd because the status is monotonic. Hence the slot we are claiming is available indeed. From this we get $s_{\widehat{k}} = 2k = s^1$, the points-to assertion $(\mathtt{items}[\widehat{k}] \leadsto_{\text{na}} -) @ \mathcal{S}_{\widehat{k}}$ and the read token of rank $k - C$. The sets of read tokens and write tokens depend on the value of `tail` and `head`, and we have just incremented the latter, to $k + 1$, so we destruct the read token of rank $k - C$ and create a write token of rank k instead, giving it as payload the item $(v, \mathcal{V})$ that we are trying to enqueue.

This is also when the *strict* monotonicity of the status comes into play: because $s_k = s^1$, it gives us $\mathcal{S}_k \sqsubseteq \mathcal{S}^1$. But we have $\uparrow \mathcal{S}^1$ in our proof context, so we obtain the points-to assertion as a subjective assertion: $\texttt{items}[\widehat{k}] \rightsquigarrow_{\text{na}} -$.

We commit the atomic update now. Indeed the successful CAS is the commit point of `try_enqueue`. We know that the program will return `true`, so we must provide the corresponding postcondition of the logically atomic triple, where our item $(v, \mathcal{V})$ has been appended to the public state of the queue. Thus we take a ghost step to update this public state. By committing, we finally obtain the assertion Ψ `true` that will serve at the end of the proof, since `true` is the return value of the operation. Along the way, we also collect the persistent assertion $\uparrow \mathcal{V}$ from the precondition of the logically atomic triple.

Finally, we keep on our side the write token, the non-atomic points-to assertion $\texttt{items}[\widehat{k}] \rightsquigarrow_{\text{na}} -$, we reconstruct the invariant, updated for the new value of `head`, and we close it.

The next step of the program writes the value v to the non-atomic item field, which is easy since we have the points-to assertion at hand. This assertion then becomes $\texttt{items}[\widehat{k}] \rightsquigarrow_{\text{na}} v$. We turn it back to an objective assertion, which gives us a view $\mathcal{W}$ and two assertions $\uparrow \mathcal{W}$ and $(\texttt{items}[\widehat{k}] \rightsquigarrow_{\text{na}} v) \mathbin{@} \mathcal{W}$.

The last step of the program is to update the (atomic) status of the slot. Once more we open the invariant. If we again note h and t the current values of `head` and `tail` (potentially different from the last time we opened the invariant), then owning a write token for rank k teaches us that $t \le k < h$. The invariant then gives us for slot $\widehat{k}$:

$$\text{Occupied } k \ (s_{\widehat{k}}, \mathcal{S}_{\widehat{k}}) \ (v_k, \mathcal{V}_k) \ \vee \ s_{\widehat{k}} = 2k$$

The left disjunct would own our write token, but we already have it and it is exclusive; hence we are in the right disjunct, $s_{\widehat{k}} = 2k = s^1$. We perform the atomic write with value $s^2 \triangleq 2k + 1$ (strict monotonicity is respected), and since we have both $\uparrow \mathcal{V}$ and $\uparrow \mathcal{W}$ in context, we can push the view $\mathcal{S}^2 = \mathcal{V} \sqcup \mathcal{W}$ to this atomic location while writing. We then switch to the left disjunct, by constituting the assertion:

$$\text{Occupied } k \ \left(s^2, \mathcal{S}^2\right) \ (v, \mathcal{V}) \quad \dashv\vdash \quad \ast\begin{cases} s^2 = 2k + 1 \\ (\texttt{items}[\widehat{k}] \rightsquigarrow_{\text{na}} v) \mathbin{@} \mathcal{S}^2 \\ \text{TokenW } \gamma_{\text{tokens}} \ k \ (v, \mathcal{V}) \\ \mathcal{V} \sqsubseteq \mathcal{S}^2 \end{cases}$$

Hence we return the non-atomic points-to assertion and the write token to the invariant before closing it.

5 A SIMPLE PIPELINE

We now demonstrate the use of our specification of a concurrent queue by a simple client application, that chains two treatments on a sequence of data, where each treatment is applied in a separate thread. Thus the sequence of intermediate values is transferred from a producer to a consumer using a concurrent queue.

5.1 Implementation of the Pipeline

The code of the application is presented in Figure 11. It provides a single function, `pipeline`, which takes as arguments two functions g and f, a sequence `xs`, and returns a sequence obtained by applying $g \circ f$ to each item of the input sequence. The functions g and f need not be pure: they can have side effects and rely on some state.

```
let pipeline g f xs =               let pipef n xs ys f =              let pipeg n ys zs g =
    let n = length xs in                for k from 0 to n − 1 do           for k from 0 to n − 1 do
    let ys = make () in                     let x = xs[k]_na in                let y = dequeue ys in
    fork (pipef n xs ys f);                 let y = f x in                     let z = g y in
    let zs = array_na[n] () in              enqueue ys y                       zs[i]_na ← z
    pipeg n ys zs g;                    done                               done
    zs
```

Fig. 11. Implementation of a pipeline

$$\left\{ |\mathtt{xs}| = n \ast \mathtt{xs} \rightsquigarrow^{*}_{\mathrm{na}} [x_0, \ldots, x_{n-1}] \ast \mathop{\Large\ast}_{0 \le k < n} \mathrm{wp}\ f\ x_k\ \{\lambda y.\, \mathrm{wp}\ g\ y\ \{\lambda z.\, R\ k\ z\}\} \right\}$$

$$\mathtt{pipeline}\ g\ f\ \mathtt{xs}$$

$$\left\{ \lambda \mathtt{zs}.\, \exists z_0, \ldots, z_{n-1}.\, |\mathtt{zs}| = n \ast \mathtt{zs} \rightsquigarrow^{*}_{\mathrm{na}} [z_0, \ldots, z_{n-1}] \ast \mathop{\Large\ast}_{0 \le k < n} R\ k\ z_k \right\}$$

Fig. 12. A specification for the pipeline

For simplicity, input and output sequences are encoded as (non-atomic) arrays, whose length can be obtained via a primitive operation `length` −. However the implementation can be modified to consume and produce lists of unknown length, or even infinite streams, provided an encoding of such data structures; we would then use a sentinel value in the queue to signal the end of stream.

The code is straightforward: we create a concurrent queue `ys`, then we fork a thread. The queue is shared between the main thread and the forked thread, while `xs` is transmitted to the forked thread. The forked thread reads items from `xs` in turn, applies f to them and enqueues the results. The main thread creates a new (non-atomic) array `zs` to store the output; then, it dequeues n items, where n is the number of items in the input sequence, applies g to them, and adds the results to `zs`. Finally, it returns `zs`.

5.2 Specification of the Pipeline

A possible specification for this pipeline is shown in Figure 12. It is higher-order, and expressed using the weakest-precondition predicate.

In postcondition we obtain one assertion $R\ k\ z_k$ for each item of the stream, related to its position k (in particular, R may relate the output value z_k to the input value x_k).

In the precondition, essentially, we want to state that for some predicates P and Q, we have the assertions $P\ k\ x_k$ for all items of the input stream, and functions f and g satisfy Hoare triples of the form:

$$\{P\ k\ x\}\ f\ x\ \{\lambda y.\, Q\ k\ y\}$$
$$\{Q\ k\ y\}\ g\ y\ \{\lambda z.\, R\ k\ z\}$$

so that, by the chaining rule, the composition satisfies:

$$\{P\ k\ x\}\ g\ (f\ x)\ \{\lambda z.\, R\ k\ z\}$$

By using weakest preconditions instead of Hoare triples, we avoid mentioning the predicate P; by taking advantage of the higher-order nature of Iris, we can nest the triples so as to conceal the intermediate predicate Q.

$$
\begin{array}{rcl}
\gamma & : & \textsc{Auth}(\textsc{Ex}(\textsc{View} \times \textsc{View} \times \textsc{List}(\textsc{Val} \times \textsc{View}))) \\
\gamma_{\mathrm{f}},\ \gamma_{\mathrm{g}} & : & \textsc{Auth}(\textsc{Ex}(\mathbb{N})) \\
\mathsf{PipeInv}\ g\ f\ R & \triangleq & \exists q, \gamma, \gamma_{\mathrm{f}}, \gamma_{\mathrm{g}}.\ \mathsf{QueueInv}\ q\ \gamma\ *\ \boxed{\mathsf{PipeInvInner}\ g\ f\ R\ \gamma\ \gamma_{\mathrm{f}}\ \gamma_{\mathrm{g}}} \\
\mathsf{PipeInvInner}\ g\ f\ R\ \gamma\ \gamma_{\mathrm{f}}\ \gamma_{\mathrm{g}} & \triangleq & \left\{ \begin{array}{l} \exists n_{\mathrm{f}}, n_{\mathrm{g}}, \mathcal{T}, \mathcal{H}, (y_{n_{\mathrm{g}}}, \mathcal{V}_{n_{\mathrm{g}}}), \ldots, (y_{n_{\mathrm{f}}-1}, \mathcal{V}_{n_{\mathrm{f}}-1}). \\ \quad \mathop{\ast} \left\{ \begin{array}{l} n_{\mathrm{g}} \le n_{\mathrm{f}}\ *\ \boxed{\bullet\, n_{\mathrm{f}}}^{\gamma_{\mathrm{f}}}\ *\ \boxed{\bullet\, n_{\mathrm{g}}}^{\gamma_{\mathrm{g}}} \\ \mathsf{IsQueue}\ \gamma\ \mathcal{T}\ \mathcal{H}\ \left[(y_{n_{\mathrm{g}}}, \mathcal{V}_{n_{\mathrm{g}}}), \ldots, (y_{n_{\mathrm{f}}-1}, \mathcal{V}_{n_{\mathrm{f}}-1})\right] \\ \mathop{\Huge\ast}_{n_{\mathrm{g}} \le k < n_{\mathrm{f}}} \left(\mathsf{wp}\ g\ y_k\ \{\lambda z.\, R\ k\ z\}\right) @\ \mathcal{V}_k \end{array} \right. \end{array} \right. \\
\mathsf{PipeF}\ g\ f\ R\ \gamma\ \gamma_{\mathrm{f}}\ n_{\mathrm{f}}\ \mathsf{xs}\ [x_0, \ldots, x_{n-1}] & \triangleq & \mathop{\ast} \left\{ \begin{array}{l} n_{\mathrm{f}} \le n\ *\ \boxed{\bullet\, n_{\mathrm{f}}}^{\gamma_{\mathrm{f}}} \\ \mathsf{xs} \leadsto^{*}_{\mathrm{na}} [x_0, \ldots, x_{n-1}] \\ \mathop{\Huge\ast}_{n_{\mathrm{f}} \le k < n} \mathsf{wp}\ f\ x_k\ \left\{\lambda y.\ \mathsf{wp}\ g\ y\ \{\lambda z.\, R\ k\ z\}\right\} \end{array} \right. \\
\mathsf{PipeG}\ g\ f\ R\ \gamma\ \gamma_{\mathrm{g}}\ n_{\mathrm{g}}\ \mathsf{zs}\ [z_0, \ldots, z_{n-1}] & \triangleq & \mathop{\ast} \left\{ \begin{array}{l} n_{\mathrm{g}} \le n\ *\ \boxed{\bullet\, n_{\mathrm{g}}}^{\gamma_{\mathrm{g}}} \\ \mathsf{zs} \leadsto^{*}_{\mathrm{na}} [z_0, \ldots, z_{n-1}] \\ \mathop{\Huge\ast}_{0 \le k < n_{\mathrm{g}}} R\ k\ z_k \end{array} \right.
\end{array}
$$

Fig. 13. Internal invariants of the pipeline

We use the primitive Cosmo assertion $|\mathsf{xs}| = n$ to stipulate that the length of a given array is n (recall that the array-points-to assertion is shorthand for a separating conjunction of points-to assertions for cells of the array in some range of indices, but it does not state that there are no cells beyond those mentioned).

5.3 Proof of the Specification for the Pipeline

We know prove the following result.

Theorem 5.1. *The code shown in Figure 11 satisfies the specification appearing in Figure 12.*

The proof relies on the assertions presented in Figure 13. The persistent assertion $\mathsf{PipeInv}\ g\ f\ R$ join together the internal invariant of the queue with that of the pipeline. The two assertions $\mathsf{PipeF}\ g\ f\ R\ \gamma\ \gamma_{\mathrm{f}}\ n_{\mathrm{f}}\ \mathsf{xs}\ [x_0, \ldots, x_{n-1}]$ and $\mathsf{PipeG}\ g\ f\ R\ \gamma\ \gamma_{\mathrm{g}}\ n_{\mathrm{g}}\ \mathsf{zs}\ [z_0, \ldots, z_{n-1}]$ are owned by the threads which compute f and g, respectively, and describe their loop invariants.

We again associate the queue to a ghost variable γ. In addition, we use two ghost variables γ_{f} and γ_{g} whose values are the current positions n_{f} and n_{g} of the loops computing f and g, respectively. This ghost state allows both threads, while in their respecting loops, to agree with the shared invariant on these values. At any time, we have $0 \le n_{\mathrm{g}} \le n_{\mathrm{f}} \le n$.

- Indices in the range $[n_{\mathrm{f}}, n)$ have not been processed by f yet. Hence, for these indices, we still have the weakest-precondition assertions $\mathsf{wp}\ f\ x_k\ \{-\}$ initially provided to the pipeline. These assertions can be regarded as a permission to run f once on the corresponding items. The thread computing f owns these assertions, and it also owns the array xs containing the input items.
- Indices in the range $[n_{\mathrm{g}}, n_{\mathrm{f}})$ have been processed by f but are yet to be processed by g. Hence, we have consumed their initial weakest-precondition assertions, and have obtained weakest-precondition assertions $\mathsf{wp}\ g\ y_k\ \{-\}$ as a result. These assertions are stored in the shared invariant. The invariant also owns the queue ys, whose contents are the intermediate items for exactly this range of indices.

- Indices in the range $[0, n_g)$ have been processed by both f and g. Hence, we have consumed their intermediate weakest-precondition assertions, and have obtained postconditions $R\ k\ z_k$ as a result. These are owned by the thread computing g, along with the array zs which contains the output items.

The key point is that the assertion $\mathsf{wp}\ g\ y_k\ \{-\}$ has been given by the thread computing f to the invariant when enqueuing the corresponding item, and will be taken by the thread computing g when dequeuing that item. This assertion is thus exchanged between two threads with differing views of the shared memory, and transits via a neutral ground: an objective invariant. We make the assertion objective by specifying at which view it holds: namely, the view $\mathcal{V}_k$ which the enqueuer had and which the dequeuer will acquire. Therefore, we rely crucially on the enqueuer-to-dequeuer synchronization guaranteed by the queue.

With these invariant assertions correctly stated, the proof is rather straightforward. When creating the pipeline, we have $0 = n_g = n_f$ and assertions PipeInvInner and PipeG hold trivially (the queue is empty and there are no output items computed yet); the assertion PipeF is constituted exactly from the preconditions of the pipeline (Figure 12), and is given by the main thread to the child thread that will compute f. When the pipeline has completed its work, we have $n_g = n_f = n$ and the assertion PipeG provides exactly the postcondition of the pipeline.

Thanks to the logically atomic triples in the specification of the queue, when enqueuing (respectively, dequeuing), we can open the invariant of the pipeline and move assertions to it (respectively, from it) as already explained.

6 RELATED WORK

The verification of fine-grained concurrent data structures is a well-studied problem with a particularly rich literature. Several approaches were tried, targeting various verification frameworks, various data structures in different contexts.

The notion of *linearizability* is central for specifying such libraries. Dongol and Derrick [2015] gave a survey of the different techniques used for linearizability of concurrent libraries at that time. Of particular interest in the context of separation logic is the technique of *logical atomicity*, which has been recently proved to be equivalent to linearizability [Birkedal et al. 2021] in the context of a sequentially consistent model. This concept has been developed through several iterations over the last decade [da Rocha Pinto et al. 2014; Jung et al. 2015; Jacobs and Piessens 2011; Svendsen et al. 2013; Jung et al. 2020]. In the present work, we adapt an unpublished, modern version of Iris's logically atomic triples [Jung 2019], to the setting of Cosmo. This is, to the best of our knowledge, the first use of logical atomicity in a weakly consistent setting.

Another popular approach for proving the correctness of concurrent libraries is the use of refinement with respect to a simpler implementation. This is the track chosen by ReLoC [Frumin et al. 2018], which has recently been combined with logical atomicity [Frumin et al. 2020]. Interestingly, ReloC has been recently used for proving the correctness of several concurrent queue implementations [Vindum and Birkedal 2021; Vindum et al. 2021], one of which is very close to ours. However, these proofs do not handle relaxed memory behaviors, so that they do not provide a solution to the problem of specifying the lack of happens-before relationship between some data structure accesses, which we discussed in §2.3. Because it lacks some happens-before relationships, our queue implementation *is not* a refinement of a naive implementation which would use a lock to protect a sequential implementation, so a refinement-based approach would not be useful for proving our library correct. The refinement approach has also been used to prove correct some data structures used in a concurrent garbage collector [Zakowski et al. 2018].

In a weakly consistent setting, new problems arise. As discussed in §2.3, even the definition of linearizability needs special care. Smith et al. [2019] propose new definitions of linearizability for the case of weak memory models. In contrast, other authors have developed new kinds of specifications which allow for weak behaviors of the library itself [Krishna et al. 2020; Emmi and Enea 2019; Raad et al. 2019]. We found that our method of combining logically atomic triples with views is expressive and allows for concise specifications at the same time. Previous works include the generalization of various methods to weak consistency: Lê et al. [2013b,a] used manual methods directly tied to the axiomatic memory model to prove the correctness of a queue and of a work-stealing algorithm, while Lahav and Vafeiadis [2015] adapted the Owicki-Gries methodology to the release-acquire fragment of the C11 memory model and applied it to verify a read-copy-update library. The idea of a separation logic for programs with a relaxed memory semantics has been first developed in the RSL logic [Vafeiadis and Narayan 2013], and further developed in subsequent work [Turon et al. 2014; Doko and Vafeiadis 2016; Kaiser et al. 2017; Dang et al. 2020; Mével et al. 2020]. None of these papers addressed the problem of the full functional correctness of a data structure. In particular, the specification proposed for a circular buffer in GPS [Turon et al. 2014] is a weak specification in the style of the persistent specification given at the end of §2.2: in contrast to ours, it does not specify in which order the elements leave the queue.

REFERENCES

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *Principles of Programming Languages (POPL)*. 55–66. https://doi.org/10.1145/1926385.1926394

John Bender and Jens Palsberg. 2019. A formalization of Java's concurrent access modes. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 142:1–142:28. https://doi.org/10.1145/3360568

Lars Birkedal, Thomas Dinsdale-Young, Armaël Guéneau, Guilhem Jaber, Kasper Svendsen, and Nikos Tzevelekos. 2021. Theorems for Free from Separation Logic Specifications. *PACM on Programming Languages* 5, ICFP (2021). https://doi.org/10.1145/3473586

Stephen Brookes and Peter W. O'Hearn. 2016. Concurrent separation logic. *SIGLOG News* 3, 3 (2016), 47–65. https://doi.org/10.1145/2984450.2984457

Stephen D. Brookes. 2004. A Semantics for Concurrent Separation Logic. In *International Conference on Concurrency Theory (CONCUR) (Lecture Notes in Computer Science, Vol. 3170)*. Springer, 16–34. https://doi.org/10.1007/978-3-540-28644-8_2

Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A logic for time and data abstraction. In *European Conference on Object-Oriented Programming*. Springer, 207–231. https://doi.org/10.1007/978-3-662-44202-9_9

Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2020. RustBelt meets relaxed memory. *Proceedings of the ACM on Programming Languages* 4, POPL (2020), 34:1–34:29. https://doi.org/10.1145/3371102

Marko Doko and Viktor Vafeiadis. 2016. A Program Logic for C11 Memory Fences. In *Verification, Model Checking and Abstract Interpretation (VMCAI) (Lecture Notes in Computer Science, Vol. 9583)*. Springer, 413–430. https://doi.org/10.1007/978-3-662-49122-5_20

Stephan Dolan, Anil Madhavapeddy, and KC Sivaramakrishnan. 2020. Multicore OCaml. https://github.com/ocaml-multicore/ocaml-multicore/wiki

Stephen Dolan, K. C. Sivaramakrishnan, and Anil Madhavapeddy. 2018. Bounding data races in space and time. In *Programming Language Design and Implementation (PLDI)*. 242–255. https://doi.org/10.1145/3192366.3192421

Brijesh Dongol and John Derrick. 2015. Verifying linearisability: A comparative survey. *ACM Computing Surveys (CSUR)* 48, 2 (2015), 1–43. https://doi.org/10.1145/2796550

Michael Emmi and Constantin Enea. 2019. Weak-consistency specification via visibility relaxation. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 60:1–60:28. https://doi.org/10.1145/3290373

Keir Fraser. 2004. *Practical lock-freedom.* Technical Report. University of Cambridge, Computer Laboratory.

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*. 442–451. https://doi.org/10.1145/3209108.3209174

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2020. ReLoC Reloaded: A Mechanized Relational Logic for Fine-Grained Concurrency and Logical Atomicity. *arXiv preprint arXiv:2006.13635* (2020).

Iris developers and contributors. 2021. Iris examples. https://gitlab.mpi-sws.org/iris/examples

Bart Jacobs and Frank Piessens. 2011. Expressive modular fine-grained concurrency specification. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 271–282. https://doi.org/10.1145/1926385.1926417

Ralf Jung. 2019. Logical Atomicity in Iris: the Good, the Bad, and the Ugly. Iris Workshop. https://people.mpi-sws.org/~jung/iris/talk-iris2019.pdf

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

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. *Proceedings of the ACM on Programming Languages* 4, POPL (Jan. 2020), 45:1–45:32. https://doi.org/10.1145/3371113

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 *Principles of Programming Languages (POPL)*. 637–650. https://doi.org/10.1145/2775051.2676980

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 *European Conference on Object-Oriented Programming (ECOOP)*. 17:1–17:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.17

Siddharth Krishna, Michael Emmi, Constantin Enea, and Dejan Jovanovic. 2020. Verifying Visibility-Based Weak Consistency. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 12075)*. Springer, 280–307. https://doi.org/10.1007/978-3-030-44914-8_11

Ori Lahav and Viktor Vafeiadis. 2015. Owicki-Gries reasoning for weak memory models. In *International Colloquium on Automata, Languages, and Programming*. Springer, 311–323. https://doi.org/10.1007/978-3-662-47666-6_25

Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing sequential consistency in C/C++11. In *Programming Language Design and Implementation (PLDI)*. 618–632. https://doi.org/10.1145/3062341.3062352

Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Computers* 28, 9 (1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

Nhat Minh Lê, Adrien Guatto, Albert Cohen, and Antoniu Pop. 2013a. Correct and efficient bounded FIFO queues. In *2013 25th International Symposium on Computer Architecture and High Performance Computing*. IEEE, 144–151. https://doi.org/10.1109/SBAC-PAD.2013.8

Nhat Minh Lê, Antoniu Pop, Albert Cohen, and Francesco Zappa Nardelli. 2013b. Correct and efficient work-stealing for weak memory models. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 69–80. https://doi.org/10.1145/2442516.2442524

Andreas Lochbihler. 2012. Java and the Java Memory Model – A Unified, Machine-Checked Formalisation. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 7211)*. Springer, 497–517. https://doi.org/10.1007/978-3-642-28869-2_25

Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java memory model. In *Principles of Programming Languages (POPL)*. 378–391. https://doi.org/10.1145/1047659.1040336

Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2020. Cosmo: A Concurrent Separation Logic for Multicore OCaml. *Proceedings of the ACM on Programming Languages* 4, ICFP (June 2020). https://doi.org/10.1145/3408978

Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2021. Coq proofs for Cosmo and examples. https://gitlab.inria.fr/gmevel/cosmo.

Peter W. O'Hearn. 2007. Resources, Concurrency and Local Reasoning. *Theoretical Computer Science* 375, 1–3 (May 2007), 271–307. https://doi.org/10.1016/j.tcs.2006.12.035

Matthew J. Parkinson, Richard Bornat, and Peter W. O'Hearn. 2007. Modular verification of a non-blocking stack. In *Principles of Programming Languages (POPL)*. 297–302. https://doi.org/10.1145/1190216.1190261

Azalea Raad, Marko Doko, Lovro Rozic, Ori Lahav, and Viktor Vafeiadis. 2019. On library correctness under weak memory consistency: specifying and verifying concurrent libraries under declarative consistency models. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 68:1–68:31. https://doi.org/10.1145/3290381

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Logic in Computer Science (LICS)*. 55–74. https://doi.org/10.1109/LICS.2002.1029817

Erik Rigtorp. 2021. MPMCQueue. https://github.com/rigtorp/MPMCQueue.

Graeme Smith, Kirsten Winter, and Robert J Colvin. 2019. Linearizability on hardware weak memory models. *Formal Aspects of Computing* (2019), 1–32. https://doi.org/10.1007/s00165-019-00499-8

Kasper Svendsen, Lars Birkedal, and Matthew J. Parkinson. 2013. Modular Reasoning about Separation of Concurrent Data Structures. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 7792)*. Springer, 169–188. https://doi.org/10.1007/978-3-642-37036-6_11

The Coq development team. 2020. *The Coq Proof Assistant*. http://coq.inria.fr/

Amin Timany and Lars Birkedal. 2021. Reasoning about Monotonicity in Separation Logic. In *Certified Programs and Proofs (CPP)*. 91–104. https://doi.org/10.1145/3437992.3439931

Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: navigating weak memory with ghosts, protocols, and separation. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 691–707. https://doi.org/10.1145/2714064.2660243

Viktor Vafeiadis and Chinmay Narayan. 2013. Relaxed separation logic: a program logic for C11 concurrency. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 867–884. https://doi.org/10.1145/2544173.2509532

Simon Friis Vindum and Lars Birkedal. 2021. Contextual refinement of the Michael-Scott queue. In *Certified Programs and Proofs (CPP)*. 76–90. https://doi.org/10.1145/3437992.3439930

Simon Friis Vindum, Dan Frumin, and Lars Birkedal. 2021. Mechanized Verification of a Fine-Grained Concurrent Queue from Facebook's Folly Library. (March 2021). https://cs.au.dk/~birke/papers/mpmc-queue.pdf Submitted for publication.

Yannick Zakowski, David Cachera, Delphine Demange, and David Pichardie. 2018. Verified compilation of linearizable data structures: mechanizing rely guarantee for semantic refinement. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. 1881–1890. https://doi.org/10.1145/3167132.3167333