Verifying a concurrent data-structure from the Dartino Framework in Iris

Morten Krogh-Jespersen
Aarhus University
Department of Computer Science
mkj@cs.au.dk

Thomas Dinsdale-Young
Aarhus University
Department of Computer Science
tyoung@cs.au.dk

Lars Birkedal
Aarhus University
Department of Computer Science
birkedal@cs.au.dk

Abstract

We specify and verify a concurrent queue data structure used in the scheduler of a real-world virtual machine, Google's Dartino Framework.

Our specification treats the queue operations as abstractly atomic. This means that a client can reason about them as if they take effect at a single instant in time, and thus impose its own invariants on the queue. The specifications also involve resource transfer: to enqueue a process, a thread transfers ownership of its descriptor to the queue.

We show that an implementation of the data structure, directly translated from the Dartino Framework source, satisfies our specification in Iris, a state-of-the-art higher-order concurrent separation logic, capable of expressing both abstract atomicity and resource transfer. Our verification is formalised in the Coq proof assistant. Hence, our work shows that Iris is both expressive and practical enough to formally reason about production code taken from "the wild".

CCS Concepts •**Software and its engineering** → **General programming languages;** •**Theory of computation** → *Program analysis;*

1 Introduction

The scheduler is the beating heart of any virtual machine – it is responsible for running and pausing processes of the system. Therefore, the scheduler must be both correct and efficient. The Dartino Framework is a virtual machine for the Dart language, which was designed by Google to run efficiently on limited hardware (such as embedded systems or IoT-devices). The work presented here is the result of a collaboration with the Google Dartino team to verify the queue data structure underlying the Dartino Framework's scheduler.

Conference'17, Washington, DC, USA

DOI: 10.1145/nnnnnnn.nnnnnnn

The Dartino Framework uses a pool of low-level (hardware) *threads* to run high-level Dart *processes*. Each thread has its own process queue, implemented as a doubly-linked list which we refer to as a *Dartino Queue*. Having a queue per thread serves to reduce contention, although threads may access the queues of other threads. For instance, a thread with no processes may steal one from another thread. In addition to the usual enqueue and dequeue operations, the data structure allows a specific process to be removed from a queue. This allows the scheduler to prioritise certain processes – for instance, to immediately schedule a process that is the recipient of a message.

Verification of sequential implementations of doubly-linked lists using shape analysis or separation logics has already been studied in detail, *e.g.* in the seminal work by Reynolds [7, 8]. Specifying and verifying the Dartino Queue is complicated by a number of factors.

Firstly, this Dartino Queue allows for concurrent access by multiple threads. We therefore require a specification that accounts for this. *Abstract atomicity* achieves this by specifying that an operation (such as enqueuing a process) appears to take effect at a single instant in time. A client can then reason about abstractly atomic operations in a simple manner, for instance by imposing new invariants on how the queue is used. Linearizability [3] is a well-known verification condition for abstract atomicity. Recently, notions of abstract atomicity have been introduced to separation logics such as TaDA [2].

Secondly, during its lifetime, a process may belong to multiple queues. This means that ownership of a process descriptor is transferred whenever it is enqueued or dequeued. This ownership transfer does not necessarily take place at the same instant that the operation atomically takes effect. Separation logics are well-equipped to reason about resource transfer; consequently, a separation logic which supports abstract atomicity is appropriate for this verification problem.

We have chosen to verify the Dartino Queue in Iris [4, 5], a state-of-the-art concurrent higher-order separation logic, implemented in the Coq proof assistant [1]. The reason for this is that the Iris Proof Mode enables us to do interactive proofs directly in Coq [6] and, moreover, Iris allows us to prove so-called atomic triples [2], which capture abstract atomicity. We can therefore give strong specifications that integrate ownership transfer and abstract atomicity.

Our case study applies Iris to verifying real-world code with non-trivial specifications. Our case study demonstrates the practicality and effectiveness of the following:

- Using resources in Iris to reason about dynamic allocation and stealing of processes which may be transferred between queues.
- Using logical atomicity in Iris in concert with resource transfer to verify strong specifications that accurately capture the intention for the real-world code.
- Using the Iris Proof Mode for formal, mechanised verification of code.

Outline. First, we describe the Dartino Queue and show the translation from C++ to Iris in §2. In §4 we give a primer to Iris and describe the invariants that will guard the Dartino Queue. In §4.4 we motivate and show stronger specifications for the operations on the Dartino Queue before we finally conclude in §6.

2 The Dartino Queue in Iris

The Dartino Framework is an experimental virtual machine, written in C++, for running the programming language Dart on devices with limited memory and processing resources. One particular goal with the Dartino Framework is to increase the computation throughput of concurrent programs that use message passing for communication. To this end, when one Dart process sends a message to another, the recipient is preferentially scheduled. This means that the Dartino Queue, which represents a process queue in the scheduler, must allow for processes that are not at the head to be removed from the queue.

In a general-purpose queue data structure, enqueuing a value typically involves allocating a new node to hold the value. For a process queue, however, the process descriptor, which exists for the lifetime of the process, directly represents a node in a queue. That is, the descriptor object holds pointers to the queue the process belongs to and its adjacent processes. This means that no allocation is necessary in enqueuing a process (which is good, since allocation is expensive and the scheduler must be as efficient as possible). On the other hand, one must handle ownership of process objects carefully, since they may belong to multiple queues during their lifetimes.

The Dartino Queue is implemented as a doubly-linked list to support removal of an arbitrary process its queue. Updating a doubly-linked list requires multiple pointer updates. To ensure that these updates occur safely in a concurrent context, the Dartino Queue uses the queue's head pointer as a spin lock.

2.1 Modelling C++ in Iris-ML

In order to verify the Dartino Queue, we translate the C++ code used by Google into Iris-ML, one of the programming languages supported by Iris. In doing so, we must faithfully represent the semantics of the original program. In particular, memory operations should have the same granularity: the ML program cannot perform an update in a single atomic step that takes multiple steps in the C++ source.

In C++, an object is represented as a contiguous block of memory holding the object's data members. A pointer to an object is the address of such a block, and members are accessed by computing offsets from the address into the block.

In Iris-ML, there are no objects, but there are references to arbitrary (untyped) values. The basic operations on references are:

- `ref v` — allocate a reference with initial value `v`;
- `!r` — atomically read the value stored in reference `r`;
- `r <- v` — atomically update the contents of reference `r` to value `v`; and
- `CAS r oldval newval` — atomically compare the contents of reference `r` with value `oldval`, updating it to `newval` if equal; return `true` if successful (the value was updated) and `false` otherwise.

One way a C++ object reference might be represented in Iris-ML is as a reference to a tuple of the object's data members. This representation is problematic, however, since any update to the object updates all of its members at once, while in C++ each data member is updated individually. Consequently, a C++ object reference is represented as a tuple of references to each of the object's data members. Each data member can thus be manipulated independently.

Apart from a reference to an object, a C++ pointer may instead hold the value `null`. To reflect that pointers are nullable in Iris-ML, we represent pointers as tagged data: `NONE` represents the null pointer, and `SOME r` represents a pointer with a valid object reference `r`. To dereference a pointer, we first apply the function `unSOME`, which strips the `SOME` tag and crashes when given `NONE`.

2.2 Doubly-Linked List with Arbitrary Removal

The interface of the Dartino Queue consists of five operations:

`makeQueue`: Construct a new Dartino Queue.
`makeProc`: Construct a new process descriptor.
`enqueue`: Append a process to a Dartino Queue.
`dequeue`: Attempt to remove the first process from a Dartino Queue, returning a pointer to the process. This can fail, returning a `null` pointer (Iris-ML: `NONE`), if the queue is empty.
`tryDequeueEntry`: Attempt to remove a specified process from a Dartino Queue. This can fail, returning `false`, if the process is no longer in the queue.

The Iris-ML implementation is given in Figure 1. We now describe each operation in detail.

New Dartino Queue. The function `mkQueue()` creates a new, empty Dartino Queue. A Dartino Queue object has

```
Definition unSOME :=
  λ: p,
     match: p with NONE => assert false
                 | SOME p' => p' end.

Definition queue_head := λ: p, Fst p.
Definition queue_tail := λ: p, Fst (Snd p).
Definition queue_sent := λ: p, Snd (Snd p).

Definition pval := λ: p, Fst p.
Definition qref := λ: p, Fst (Snd p).
Definition prev := λ: p, Fst (Snd (Snd p)).
Definition next := λ: p, Snd (Snd (Snd p)).

Definition makeQueue :=
  λ: <>,
     (ref NONE, (ref NONE, ref ())).

Definition makeProc :=
  λ: v,
     (ref v, (ref NONE, (ref NONE, ref NONE))).


Definition obtainLockDeq :=
  rec: loop head sentinel h :=
    let: hv := !h in
    if: (hv = SOME sentinel)
        || (~ CAS head hv (SOME sentinel))
    then h <- !head ;;
         if: (!h) = NONE then true
         else loop head sentinel h
    else false.

Definition dequeue :=
  λ: head tail s,
     λ:
        let: h := ref !head in
        if: !h = NONE then NONE
        else let: obtLock := obtainLockDeq head s h in
             if: obtLock then NONE
             else let: h' := unSOME (!h) in
                  let: next := !(next h') in
                  (if: next = NONE then
                     tail <- NONE
                   else prev (unSOME next) <- NONE)
                   ;; next h' <- NONE
                   ;; qref h' <- NONE
                   ;; head <- next
                   ;; SOME h'.
```

```
Definition obtainLockEnq :=
  rec: loop head sentinel h :=
    let: hv := !h in
    if: (hv = SOME sentinel) || (~ CAS head hv (SOME sentinel))
    then h <- !head ;; loop head sentinel h
    else ().

Definition enqueue :=
  λ: head tail s,
     λ: p,
        let: h := ref !head in
           obtainLockEnq head s h
        ;; qref p <- SOME (head,(tail,s))
        ;; match: !h with
              NONE => tail <- SOME p
                      ;; head <- SOME p
                      ;; true
            | SOME h' =>  prev p <- !tail
                          ;; next (unSOME !tail) <- SOME p
                          ;; tail <- SOME p
                          ;; head <- SOME h'
                          ;; false
          end.

Definition tryDequeueEntry :=
  λ: head tail s,
     λ: p,
        let: h := ref !head in
        if: !h = NONE then false
        else let: obtLock := obtainLockDeq head s h in
             if: obtLock then
               false
             else if: !(qref p) = SOME (head,(tail,s)) then
                    let: next := !(next p) in
                    let: prev := !(prev p) in
                    (if: next = NONE then
                       tail <- prev
                     else
                       prev (unSOME next) <- prev)
                    ;; (if: prev = NONE then
                          h <- next
                        else
                          next (unSOME prev) <- next)
                    ;; (prev p) <- NONE
                    ;; (next p) <- NONE
                    ;; (qref p) <- NONE
                    ;; head <- !h
                    ;; true
                  else
                    head <- !h ;; false.
```

Figure 1. Implementation of a doubly-linked queue with a virtual lock. Function binders in Iris-ML are strings in Coq, but are shown as regular binders for the sake of clarity.

three data members: the pointers head and `tail` to the head and tail of the queue, and a distinguished sentinel value `sent`. To indicate when the lock on the queue is held, the head pointer is set to the sentinel. To ensure that this sentinel value is distinct from any process reference, the `makeQueue` constructor generates a new reference (whose contents is immaterial). The head and `tail` pointers are both initialised to NONE (representing `null`). Figure 2 shows the initial configuration of a Dartino Queue object.

New Process. The function `mkProcess(v)` constructs a new process descriptor holding value `v`. (The meaning of the value is determined by the client of the queue, which in the Dartino Framework is the process's instruction pointer.) A process descriptor has four members: a value `pval`; a pointer

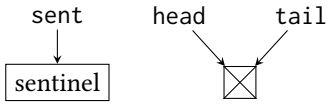
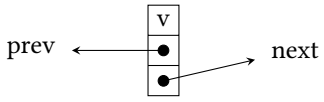


Figure 2. Initial configuration of the queue

to the queue that currently holds the process, `qref`; and pointers `prev` and `next` to the previous and next processes in the queue respectively. We depict processes as so (where the `qref` pointer is not drawn since the queue that owns the process is obvious from the context):



In Iris-ML, a process object is represented as a tuple of references, and we define four projections out of the tuple named `pval`, `qref`, `prev` and `next`.

Enqueuing. The function `enqueue(q,p)` enqueues process `p` in the Dartino Queue `q`. Enqueuing elements involves obtaining the (virtual) lock of the Dartino Queue, inserting the new element once the lock is acquired, and finally releasing the lock again. These steps are illustrated in Figure 3.

Obtaining the lock is delegated to `obtainLockEnq`, which loops attempting to update the head pointer of the queue (`head`) to the sentinel value (`sentinel`); the old value of the head pointer is recorded in the reference `h`. The function retries if the head currently holds the sentinel value (indicating that another thread holds the lock) or if the `CAS` fails as a result of another thread updating it. When `obtainLockEnq` returns, it must have successfully updated the head pointer from the (non-sentinel) value now stored in `h` to the sentinel value. Thus the thread will have acquired the lock. Obtaining the lock takes us from (a) to (b) in Figure 3.

Once the lock is held, the thread is at liberty to modify the list, and can assume that no other thread will concurrently modify it. The process is added to the end of the list by performing four pointer updates: the process's `qref` pointer is updated to point to the queue; the process's `prev` pointer is updated to point to the original `tail`; the tail's `next` pointer is updated to point to the new process; and the `tail` pointer is updated to point to the new process. This update takes us from (b) to (c) in Figure 3. In the case where the list was initially empty, it is only necessary to update the process's `qref` pointer and the queue's `tail` pointer.

To complete the enqueue operation, the `head` pointer is updated to point to the original head of the list (which was stored in `h`), if the list was non-empty. This takes us from (c) to (d) in Figure 3. If the list was empty, the `head` pointer is updated to point to the newly enqueued process.

Dequeuing. The function `dequeue(q)` dequeues the process at the head of the Dartino Queue `q`. As with enqueuing,

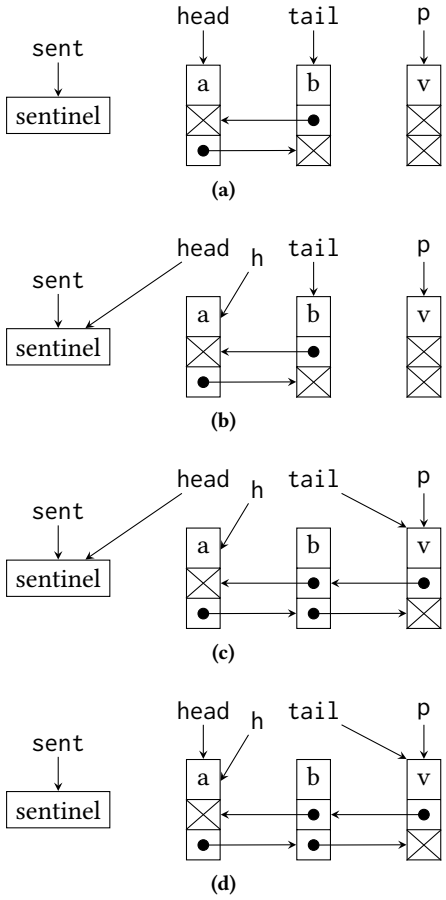


Figure 3. Enqueuing an element into the Dartino Queue.

the operation involves acquiring the lock, updating the list, and finally releasing the lock. This is depicted in Figure 4.

Before attempting to obtain the lock, a test checks if the head pointer was `NONE`, indicating that the queue was empty, in which case the function immediately returns `NONE`. Otherwise, an attempt to acquire the lock is made by calling `obtainLockDeq`.

`obtainLockDeq` behaves like `obtainLockEnq` in acquiring the lock, except that it does not attempt to acquire the lock if the queue is empty; it returns `true` if the queue was empty, and therefore the lock was not acquired, and `false` if the lock was successfully acquired with the queue non-empty.

When the lock is successfully acquired, `h` holds a (non-null) pointer to the process descriptor at the head of the queue (Figure 4 (b)). The descriptor's `next` pointer is inspected to determine if it is the end of the queue, in which case it will be `NONE`. If so, the queue's `tail` pointer is set to `NONE` since the queue will now be empty. If not, the next process's `prev` pointer is set to `NONE`, since it will now be the head of the queue. The `next` and `qref` fields of the head process are both updated to `NONE`, since it is being removed from the queue

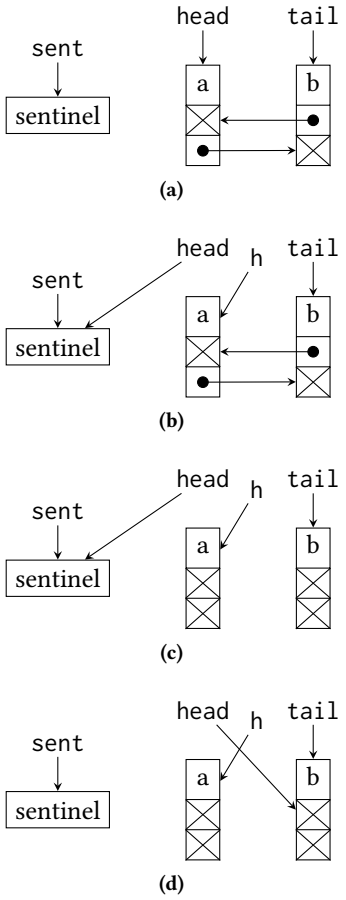


Figure 4. Dequeuing an element from the Dartino Queue.

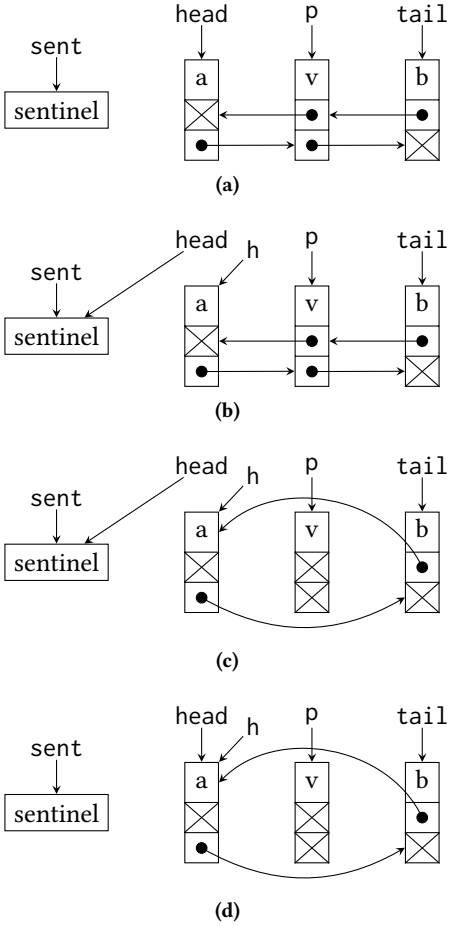


Figure 5. Dequeuing a specific element p from the Dartino Queue.

(Figure 4 (c)). Finally, the queue's head is updated to point to the new head process (the successor of the removed process, before it was removed).

Arbitrary Dequeuing. The most interesting aspect of the Dartino Queue is that a specific process p can be removed from a queue q with the function `tryDequeueEntry(q,p)`. This is shown in Figure 5.

As with dequeue, the first step is to acquire the lock for the queue, but only if the queue is non-empty. If the queue is empty then the process cannot be in the queue (perhaps another thread already dequeued it) and so `tryDequeueEntry` returns `false`. Otherwise, it is necessary to check that the process's `qref` pointer points to the queue, since even if this was initially the case, another thread may have dequeued the process since `tryDequeueEntry` was called. If `qref` does not match the queue, the lock is released by updating head to its previous value and the operation returns `false`. Otherwise, we can be sure that the process indeed belongs to the queue, and since the thread holds the lock on the queue, no other thread can concurrently dequeue it (Figure 5 (b)).

The process p is removed from the queue by first updating the `prev` pointer of its successor to the `prev` pointer of p. If the successor is `NONE` then the `tail` pointer is updated instead, since p must be the last process in the queue. Next, the `next` pointer of p's predecessor is updated to point to the `next` pointer of p. Again, if there is no predecessor, the h pointer is updated instead, since p must be the first process in the queue. Next, the `prev`, `next` and `qref` pointers of p are all set to `NONE` (Figure 5 (c)). Finally, the lock on the queue is released by updating head to the pointer stored in h, which is either the former head of the queue (if it was not p) or the successor of p (if p used to be the head).

3 The Iris Logic

We specify and verify the Dartino Queue in Iris, a concurrent higher-order separation logic implemented in Coq. Iris is built around monoids and invariants. Monoids provide a way to define abstract (ghost) resources that represent knowledge and rights available to threads. Invariants provide a way to give concrete meaning to these abstract resources.

Iris includes the following quantifiable types:

$$\kappa ::= \mathbf{1} \mid \kappa \times \kappa \mid \kappa \to \kappa \mid \mathit{Expr} \mid \mathit{Val} \mid \mathbb{B} \mid \mathbb{N} \mid \mathit{Names} \mid \mathit{Monoid} \mid \mathit{iProp} \mid \ldots$$

Here, $\mathbf{1}$, $\mathbb{B}$ and $\mathbb{N}$ is the unit type, the type of booleans and the type of natural numbers respectively. *Expr* and *Val* are syntactic expressions and values of Iris-ML. *Monoid* is the type of monoids, which are used for ghost resources. *Names* is the type of ghost names, which is used to assign names to instances of ghost resources. *iProp* is the type of Iris propositions, which are defined by the following grammar:

$$P ::= \top \mid \bot \mid P \wedge P \mid P \vee P \mid P \Rightarrow P \mid P * P \mid P \mathrel{-\!\!*} P \mid \forall x : \kappa.\ \Phi \mid \exists x : \kappa.\ \Phi \mid \triangleright P \mid \mu r.P \mid \checkmark(a) \mid \Box P \mid \models\{\mathsf{E1},\mathsf{E2}\}\Rightarrow P \mid \mathtt{own}\ \gamma\ \mathsf{a} \mid \mathtt{inv}\ \mathsf{N}\ \mathsf{P} \mid \ldots$$

The grammar includes the usual higher-order logic connectives ($\top$, $\bot$, $\wedge$, $\vee$, $\Rightarrow$, $\forall$, $\exists$). The separating conjunction $*$ describes resources that are split into two disjoint parts. Magic wand, $P \mathrel{-\!\!*} Q$, behaves like implication for resources: if resources P are given up, then the resources described by Q can be obtained. Ownership of ghost resources is written as $\mathtt{own}\ \gamma\ \mathsf{a}$ where a is a monoid element and γ is a ghost name. Owned resources must be valid, which is asserted by $\checkmark(\mathsf{a})$. The update modality $\models\{\mathsf{E1},\mathsf{E2}\}\Rightarrow$ allows resources to be updated, where the masks $\mathsf{E1}$ and $\mathsf{E2}$ describe the set of invariants that are not open before and after the update, respectively. When the masks are the same we write $\models\{\mathsf{E}\}\Rightarrow$.

The $\Box$ modality asserts that a proposition holds independently of resources. Consequently, the proposition $\Box P$ is *persistent*: it can be freely duplicated as it satisfies $\Box P \Leftrightarrow \Box P * \Box P$.

Invariants, $\mathtt{inv}\ \mathsf{N}\ \mathsf{P}$, where N is the name of the invariant and P the invariant assertion, are also persistent propositions and hence duplicable. The resources embodied by an invariant can only be accessed during atomic operations, which must reestablish the invariant. This ensures that no thread can see a violation of the invariant: it is indeed invariant.

Updating resources and opening invariants modify the ghost state without executing any code. The following two connectives, *view shift* and *wand shift,* are particularly useful for these tasks:

$$P \mathrel{=\!\!\{\mathsf{E1},\mathsf{E2}\}\!\!\Rightarrow} Q \triangleq \Box(P \mathrel{-\!\!*} \models\{\mathsf{E1},\mathsf{E2}\}\Rightarrow Q)$$

$$P \mathrel{=\!\!\{\mathsf{E1},\mathsf{E2}\}\!\!=\!\!*} Q \triangleq P \mathrel{-\!\!*} \models\{\mathsf{E1},\mathsf{E2}\}\Rightarrow Q$$

The behaviour of view shifts and wand shifts are similar to Hoare-triples, taking a precondition P, that asserts the shape of the ghost state before updating and a postcondition Q, the ghost state obtained by running the view shift or wand shift. There is not need for any code, since these shifts only update ghost resources and not the physical state. Note that the view shift is persistent: it cannot depend on any currently-available resources. By contrast, the wand shift may use up available resources in the update.

$$\textsc{Monoid-Alloc}\ \frac{\checkmark(\mathsf{a})}{\models\{\mathsf{E}\}\Rightarrow\exists\gamma,\ \mathtt{own}\ \gamma\ \mathsf{a}} \qquad \textsc{Monoid-Update}\ \frac{\mathsf{a} \rightsquigarrow \mathsf{B}}{\mathtt{own}\ \gamma\ \mathsf{a} \vdash \models\{\mathsf{E}\}\Rightarrow\exists \mathsf{b} \in \mathsf{B},\ \mathtt{own}\ \gamma\ \mathsf{b}}$$

$$\textsc{Monoid-Valid}\ \mathtt{own}\ \gamma\ \mathsf{a} \vdash \checkmark(\mathsf{a}) \qquad \textsc{Monoid-Op}\ \mathtt{own}\ \gamma\ \mathsf{a} * \mathtt{own}\ \gamma\ \mathsf{b} \dashv\vdash \mathtt{own}\ \gamma\ \mathsf{a} \cdot \mathsf{b}$$

Figure 6. Rules for monoid resources in Iris

3.1 Monoids

Commutative monoids are the bread and butter of any separation logic. A commutative monoid consists of a set with a binary operation (which we denote $\cdot$) that is associative, commutative and has an identity element. In Iris, arbitrary monoids can be used as ghost resources. (Technically, Iris uses *resource algebras*, which relax the identity property, but have some other properties. We will abuse the terminology by referring to resource algebras as monoids even when they do not have an identity element.)

Figure 6 shows Iris rules for working with monoid resources. The Monoid-Alloc rule allows a new ghost resource to be allocated holding any valid monoid element. The Monoid-Valid rule requires that any allocated resource must hold a valid monoid element. The Monoid-Update rule allows a ghost resource to be updated. The $\rightsquigarrow$ represents frame-preserving update: if $\mathsf{a} \rightsquigarrow \mathsf{B}$ and $\checkmark(\mathsf{a} \cdot \mathsf{c})$ then it must be that $\checkmark(\mathsf{b} \cdot \mathsf{c})$ holds for some $\mathsf{b} \in \mathsf{B}$. Frame-preserving updates thus do not invalidate the ownership of any other concurrent threads. The Monoid-Op rule allows ghost resources to be split and joined with the monoid composition operator.

We now present a number of standard monoid constructions that are useful in our verification.

Exclusive. The Ex(S) monoid (over a given set S) is one of the simplest monoids, where the composition $a \cdot b$ is undefined everywhere. ("Undefined" is represented by a distinguished element of the monoid that is not valid.) For the exclusive monoid, we thus have the following law:

```
∀ a b, own γ (Excl a) ∗ own γ (Excl b) ⊢ ⊥.
```

There cannot be two owned instances at one point in time, therefore one is always free to update the resource using the Monoid-Update rule.

Decidable Agreement. The DecAgree(S) monoid over a set S with decidable equality has composition defined by $s \cdot s = s$ for all $s \in S$, but undefined otherwise. This means that if two threads own elements of this monoid then they must agree. This is expressed by the following Iris proposition:

```
∀ a b, own γ (DecAgree a) ∗ own γ (DecAgree b) ⊢ a = b.
```

To show this holds, we use Monoid-Op to combine the components into one assertion. Then, by Monoid-Valid, we have that $a \cdot b$ is valid, but that can only be true if $a = b$. Notice that we cannot perform a frame-preserving update for this monoid.

Fractional Permissions. Given a monoid M, the fractional permissions monoid Frac(M) has carrier $(\mathbb{Q} \cap (0, 1]) \times M$. Composition is defined as $(\pi_1, a) \cdot (\pi_2, b) = (\pi_1 + \pi_2, a \cdot b)$, where the sum $\pi_1 + \pi_2$ may not exceed 1. We thus have the following Iris propositions:

```
∀ a b, own γ (π,a) ⊢ own γ (π/2,a) ∗ own γ (π/2,a).
∀ a b, own γ (π,a) ∗ own γ (π',a) ∗ π + π' <= 1
     ⊢ own γ (π + π',a).
```

If one has `own γ (1,a)`, no other fractions can be owned. Thus one has exclusive ownership and can freely update the resource.

Finite Sets. Given a set X, the finite-set monoid Gset(X) consists of the finite subsets of X under disjoint union. That is, the composition of two finite subsets $a \cdot b$ is defined as the union $a \cup b$ when $a \cap b = \emptyset$, and undefined otherwise.

Finite Maps. Given a set X and monoid Y, the finite-map monoid Gmap(X, Y) consists of the finite partial functions from X to Y. Composition is given by:

$$(a \cdot b)(x) = \begin{cases} a(x) \cdot b(x) & \text{if } x \in \mathrm{dom}(a) \text{ and } x \in \mathrm{dom}(b) \\ a(x) & \text{if } x \in \mathrm{dom}(a) \text{ and } x \notin \mathrm{dom}(b) \\ b(x) & \text{if } x \notin \mathrm{dom}(a) \text{ and } x \in \mathrm{dom}(b) \end{cases}$$

Composition is thus functorial in the co-domain monoid:

```
{[ i := x ]} · {[ i := y ]} = {[ i := x · y ]}
```

where `{[ i := x ]}` represents the singleton map from `i` to `x`.

A frame-preserving update can extend the domain of a finite map with a new key, provided that we are not specific about *which* new key:

$$\checkmark(x) \vdash m \rightsquigarrow \{m' \mid \exists i.\, m' = \{[i := x]\} \cdot m \wedge i \notin \mathrm{dom}(m)\}$$

This gives a way of allocating new ghost resources in a monoid.

Authoritative. Given a monoid X, the authoritative monoid is built from two types of resources: authoritative resources $\bullet\, a$, and fragment resources $\circ\, b$. Fragment resources can be composed according to the underlying monoid: $\circ\, a \cdot \circ\, b = \circ\,(a \cdot b)$. Authoritative resources cannot be composed with each other: $\bullet\, a \cdot \bullet\, b$ is undefined. When an authoritative and fragment resource are combined, the fragment must be contained with in the authoritative resource: $\checkmark(\bullet\, a \cdot \circ\, b)$ implies $b \preccurlyeq a$, where the induced monoid ordering $b \preccurlyeq a$ means that there exists some c such that $b \cdot c = a$.

File name	Contents
program.v	the Dartino Queue implementation
definitions.v	record definitions that model processes and queues
monoids.v	declarations of ghost resources and lemmas about them
invariants.v	invariants for processes and queues
helpers.v	helper lemmas for the invariants
wp_helpers.v	helper lemmas regarding weakest-precondition reduction of terms
atomize.v	the definition of abstract atomicity
makers.v	proofs for the queue and process constructors
enqueue.v	proofs for enqueuing processes
dequeue.v	proofs for dequeuing processes (at the head and arbitrarily)
client_sequential.v	proofs for a sequential client of the Dartino Queue
client_concurrent.v	proofs for a concurrent client of the Dartino Queue

Table 1. Organization of the Coq project.

To perform a frame-preserving update in the authoritative monoid, one typically requires the authoritative resource, and any such update must preserve all fragments that may be owned by other threads. For instance, it is possible to extend the authority by introducing a new fragment:

```
∀ γ a b, own γ •a ∗ ✓(a · b) ⊢ own γ •(a · b) · ○b.
```

The Heap We can now give the ordinary Heap monoid in terms of the above constructions:

$$\textsc{Heap} \triangleq \textsc{Auth}(\textsc{Gmap}(\mathbb{N}, \textsc{Ex}(\mathit{Val})))$$

Having ownership of an authoritative part of a heap is then `own γ • h`, where local ownership of a points-to predicate is written as `own γ ○ {[l := Excl v]}`, which we can give a nicer syntactic representation as `l ↦ v`.

4 A Specification for the Dartino Queue

In this section, we present the Iris specification for the Dartino Queue. Table 1 shows the structure of the Coq project. While the Coq development includes all proofs, we only present the specifications here.

To simplify our presentation, we take a few liberties with the Coq syntax. In particular, we omit some injections between types (*e.g.* from Coq propositions into Iris terms) as well as scope specifiers.

4.1 Datatype Definitions

A reference to a process descriptor object is modelled as a record of four locations, a *process address*. These locations correspond to the addresses of the data members of the object.

```
Record procAddrT := ProcAddrT {
  pvall : loc; pqueuel : loc; pprevl : loc; pnextl : loc
}.
```

A *queue address* is similarly defined as a record of three locations that comprise the data members of a queue object.

```
Record queueAddrT := QueueAddrT {
  qhead : loc; qtail : loc; qsent : loc
}.
```

A *process value* record models the contents of a process descriptor object. It thus comprises the values of each data member of the object.

```
Record procValT := ProcValT' {
  pvalv : val;
  pqueuev : option queueAddrT;
  pprevv : option procAddrT;
  pnextv : option procAddrT
}.
```

4.2 Monoids

Our specification of the Dartino Queue uses four custom monoids to represent ghost state.

4.2.1 Process Monoid

The process monoid is the authoritative monoid on partial maps from process addresses to process values with fractional permissions:

$$\textsc{Auth}(\textsc{Gmap}(\texttt{procAddrT}, \textsc{Frac}(\textsc{DecAgree}(\texttt{procValT}))).$$

This monoid represents the current state of process descriptor objects. The authoritative part of the monoid belongs to an invariant (described by `procs_inv`), which ensures that the `pval` pointer of each process matches the value recorded in the monoid. A ¼ fraction of the fragment part typically belongs to an invariant for the process (described by `proc_inv`), which establishes the relationship between the `qref`, `prev` and `next` pointers and the values in the monoid. The remaining ¾ fraction represents ownership of the process, which may either belong to a queue (if the process is in that queue) or a thread.

To denote a fragment part with a given fraction, we define:

```
Definition Proc (x : procAddrT)(π : Qp)(v : procValT)
  := ○ {[ x := (π, DecAgree v) ]}.
```

4.2.2 Queue Membership Monoid

The queue membership monoid is the authoritative monoid on finite sets of process addresses:

$$\textsc{Auth}(\textsc{Gset}(\texttt{procAddrT})).$$

Each queue has an instance of this monoid that tracks which processes currently belong to it. The authoritative part of the monoid belongs to the predicate that represents a queue, which maintains that the processes recorded in the monoid are exactly those belonging to the queue. The authoritative part is represented as `InQueueAuth l`, where `l` is a list of process addresses. When a process belongs to a queue, the invariant for the process holds a (singleton) fragment of the monoid to track that the process does indeed belong to the queue. This fragment is represented as `InQueue p`, where `p` is a process address. The authoritative monoid gives us the following property:

```
own γ (InQueueAuth l) ∗ own γ (InQueue a) ⊢ a ∈ l
```

4.2.3 Link Monoid

Since queues may be dynamically created, their ghost resources (*i.e.* the queue membership monoid for a queue) are also dynamically allocated. To track these, we use a link monoid that records the ghost resource name associated with queues. This monoid is the authoritative monoid on maps from locations to ghost resource names:

$$\textsc{Auth}(\textsc{Gmap}(\mathit{Loc}, \textsc{DecAgree}(\mathit{Names}))).$$

A fragment part is represented as `Link qs γ`, indicating that the queue with sentinel `qs` is associated with ghost name γ. The authoritative part belongs to a global invariant (`queues`) which tracks the current queues. The following useful property holds for the link monoid:

```
own γq (Link (qsent q) γ) ∗ own γq (Link (qsent q) γ')
⊢ γ = γ'
```

4.2.4 List Monoid

The list monoid is used to track the list of processes that logically belong to a queue. This is used to ensure that, when a thread holds the lock on a queue, no other thread can update the logical contents of the queue: the monoid records the logical contents of the queue; the thread has half of the resource and the queue has the other half; both halves must agree, so only the thread with the lock can update the queue. This monoid is the fractional monoid on lists of process addresses:

$$\textsc{Frac}(\textsc{DecAgree}(\texttt{list procAddrT})).$$

We define `List l` to be a ½ fraction with value `l`. This monoid has the following important property:

$$\texttt{own}\ \gamma\ (\texttt{List}\ \texttt{l}) * \texttt{own}\ \gamma\ (\texttt{List}\ \texttt{l}') \vdash \texttt{l} = \texttt{l}'$$

and, for updating,

$$\texttt{own}\ \gamma\ (\texttt{List}\ \texttt{l}) * \texttt{own}\ \gamma\ (\texttt{List}\ \texttt{l}) \vdash \texttt{own}\ \gamma\ (\texttt{List}\ \texttt{l}') * \texttt{own}\ \gamma\ (\texttt{List}\ \texttt{l}')$$

4.3 Predicate Definitions

We now define predicates to represent processes and queues, using the above monoids.

4.3.1 Processes

The proc predicate specifies a process:

```
Definition proc γp γq (a : procAddrT) (v : val)
 (qv : option queueAddrT)(pv nv : option procAddrT) :=
  own γp (Proc a 1/4 {| pvalv := v; pqueuev := qv;
                        pprevv := pv; pnextv := nv |}) ∗
  (pqueuel a) ↦ option_queueAddrT_to_val qv ∗
  (pprevl a) ↦ option_procAddrT_to_val pv ∗
  (pnextl a) ↦ option_procAddrT_to_val nv ∗
  match qv with
  | None => True
  | Some q => ∃ γ, own γq (Link (qsent q) γ) ∗
                   own γ (InQueue a)
  end.
```

The assertion proc γp γq a v qv pv nv declares ownership of the points-to predicates for the qref (pqueuel a), prev (pprevl a) and next (pnextl a) locations. These locations hold pointers to the specified queue (qv), previous process (pv) and next process (nv) respectively. Furthermore, the assertion declares ownership of a ¼ fragment of the corresponding process ghost resource.

If the process belongs to a queue (*i.e.* qv is Some q) then the assertion establishes this relationship by holding the ghost resources own γq (Link (qsent q) γ) and own γ (InQueue a) (for some γ). The first of these certifies that the ghost name associated with the queue is γ, while the second certifies that the process logically belongs to the queue.

The predicate proc_inv γp γq x wraps the proc predicate in an invariant (with other parameters existentially quantified). The qproc predicate combines ¾ ownership of the Proc ghost resource for a process with the proc_inv invariant:

```
Definition qproc γp γq x :=
  (∃ v, own γp (Proc x 3/4 {| pvalv := v;
                              pqueuev := None;
                              pprevv := None;
                              pnextv := None |})
      ∗ proc_inv γp γq x)%I.
```

Since the Proc fragment from the qproc predicate must agree with the Proc fragment from the proc_inv invariant, we can be sure that the heap cells representing the process object will hold the appropriate values. The qproc requires that the qref, prev and next pointers should all be None — *i.e.* the process does not belong to any queue.

4.3.2 Queues

A queue is represented by the queue γp γq q γ γ' l predicate, where q is the queue address, l is a list of the process addresses for processes that belong to the queue, and the remaining parameters are ghost resource names. A queue may either be locked or unlocked. If it is locked then the queue's head pointer must point to the sentinel value, and the majority of the resources representing the queue will have been transferred to the thread that holds the lock. If the queue is unlocked then these resources (which are represented by the queue_lock predicate) will belong to the queue. In either state, the queue maintains ½ ownership of the head and sentinel points-to predicates, since threads require access to these in both cases. The predicate also includes one (of two) List l ghost resources that tracks the logical contents of the queue; the other is in the queue_lock predicate. Finally, it includes a Link (qsent q) γ' ghost resource, which records that γ' is the ghost name for the queue's list membership resource. The predicate is defined as follows:

```
Definition queue γp γq (q : queueAddrT) γ γ' l :=
 ∃ (hv : val) (hvOP tvOP : option procAddrT),
 (qhead q) ↦{1/2} hv ∗ (qsent q) ↦{1/2}() ∗
 own γ (List l) ∗ own γq (Link (qsent q) γ') ∗
 (hv = SOMEV (qsent q) ∨
   (hv = (option_procAddrT_to_val hvOP) ∗
    queue_lock γp γq q γ γ' hv hvOP tvOP l)).
```

The queue_lock γp γq q γ γ' hv hv' tv l predicate represents the majority of the resources that constitute a queue, and which may be obtained by a thread on acquiring the lock. Here, q is the queue address, hv is the value of the queue's head pointer, hv' and tv are pointers to the head and tail processes in the queue respectively, and l is a list of process addresses that are in the queue.

```
Definition queue_lock γp γq (q : queueAddrT) γ γ'
  (hv : val) (hv' tv : option procAddrT)
  (l : list procAddrT) := (qhead q) ↦{1/2} hv ∗
    own γ (List l) ∗ own γ' (InQueueAuth l) ∗
    (qtail q) ↦ option_procAddrT_to_val tv ∗
    queue_cont γp γq q hv' tv l.
```

The queue_lock predicate includes half ownership of the queue's head pointer and the second List l ghost resource for the queue. These resources complement those of the queue predicate, and ensure that the head pointer and logical contents of the queue cannot be changed by other threads while the lock is held.

The predicate also asserts ownership of InQueueAuth l ghost resource, which ensures that only processes in l can have a corresponding InQueue resource. Moreover, the full permission on the queue's tail pointer belongs to the queue_lock predicate, since this pointer is only accessed by threads that have acquired the lock. Finally, the list of processes, represented by the queue_cont predicate, completes the predicate.

The predicate queue_cont consists of proc_inv invariants for each process in the list, together with a recursively-defined predicate queue_dll that ensures that the processes form a doubly-linked list.

```
Definition queue_cont γp γq (q : queueAddrT)
  (h t : option procAddrT) (l : list procAddrT) :=
```

```
    ([* list] p ∈ l, proc_inv γp γq p) ∗
    queue_dll γp q l h t None None.
```

The `queue_dll γp q l i e p n` resembles a standard doubly-linked list segment predicate [8], except that `Proc` ghost resources are used to represent the nodes of the list. (The `proc_inv` invariant for each process establishes the connection between these ghost resources and the actual values in the process object, since it holds the complementary `Proc` resource.) The pointers `i` and `e` are to the first and last processes in the segment, respectively, and `p` and `n` are the previous and next pointers of the first and last nodes of the segment.

```
Fixpoint queue_dll γp (q : queueAddrT) (l : list
  procAddrT) (i e p n : option procAddrT) :=
  match l with
  | nil => (i = n ∧ e = p)
  | x :: l' => ∃ (v : val) (n' : option procAddrT),
               i = Some x ∗
               own γp (Proc x ¾ {| pvalv := v;
                                   pqueuev := (Some q);
                                   pprevv := p;
                                   pnextv := n' |}) ∗
               queue_dll γp q l' n' e i n
  end.
```

4.4 A Logically Atomic Specification for the Dartino Queue

One approach to specifying the Dartino Queue would be with Hoare-triples such as the following[1]:

$$\{\mathsf{qproc\ p} * \mathsf{queue\ q\ l}\}$$
$$\mathsf{enqueue(q, p)}$$
$$\{\mathsf{v.\ v} = () * \mathsf{queue\ q\ (l\ ++\ [p])}\}$$

This specifies that calling enqueue with a valid queue `q` and un-enqueued process `p` will result in the process being appended to the queue. Unfortunately, to use this specification, a thread must have ownership of the queue. Therefore, it is not useful in a concurrent situation where the queue may be shared among many threads (such as a scheduler).

An alternative specification would be to wrap the queue in an invariant:

$$\{\mathsf{qproc\ p} * \mathtt{inv\ N}\ (\exists \mathsf{l.\ queue\ q\ l})\}$$
$$\mathsf{enqueue(q, p)}$$
$$\{\mathsf{v.\ v} = () * \mathtt{inv\ N}\ (\exists \mathsf{l.\ queue\ q\ l})\}$$

With such a specification, multiple threads can access the queue. However, we lose the information that enqueue actually appends the process to the queue. Indeed an implementation could not change the queue at all and be correct with respect to such a specification.

The problem with the first specification is that we do not allow any concurrent updates to the queue. The problem with the second is that we allow all possible concurrent updates to the queue. The optimal specification would allow the client of the queue to determine exactly which concurrent updates are possible. We can achieve such a specification by viewing the update as *logically atomic* [2].

Access to invariants is generally only permitted to atomic operations: if the operation preserves the invariant, then no other thread can observe a violation of the invariant because the operation is atomic. Logically atomic operations can similarly be used to access invariants, although they do not execute in a single atomic step. In [2], da Rocha Pinto *et al.* propose an *atomic triple* for specifying logical atomicity. For enqueue, we might give the following atomic triple:

$$\mathbb{\forall} \mathsf{l}.\langle \mathsf{qproc\ p} * \mathsf{queue\ q\ l}\rangle$$
$$\mathsf{enqueue(q, p)}$$
$$\langle \mathsf{v.\ v} = () * \mathsf{queue\ q\ (l\ ++\ [p])}\rangle$$

This specification expresses that the process `p` is atomically appended to the queue `q` in the execution of `enqueue(q,p)`. The binding of `l`, representing the contents of the queue, allows the client to arbitrarily update the queue during the execution of enqueue, provided that the precondition holds for *some* `l` up until the atomic update takes effect. Immediately after the atomic update, the postcondition will hold for the value of `l` at which the precondition held immediately prior.

4.4.1 Logical Atomicity in Iris

In Iris hoare-triples are encoded using weakest precondition, as so:

$$\{P\}\ e\ \{\Phi\} \triangleq \Box(P \mathrel{-\!\!*} \mathsf{wp}\ e\ \{\Phi\})$$

Therefore, we show how to construct a logically-atomic weakest precondition in Iris. The core idea is expressed by an "atomic shift" [4]:

```
Definition atomic_shift {A B : Type}
  (α: A → iProp Σ)      (* atomic pre-condition *)
  (β: A → B → iProp Σ) (* atomic post-condition *)
  (Ei Eo: coPset)       (* inside/outside invs *)
  (P : iProp Σ) (Q : A → B → iProp Σ) : iProp Σ :=
    (P ={Eo, Ei}=> ∃ x:A, α x ∗
      ((α x ={Ei, Eo}=∗ P) ∧
        (∀ y, β x y ={Ei, Eo}=∗ Q x y))).
```

An atomic shift is a persistent assertion. It effectively captures that an atomic update from α to β is sufficient to take precondition `P` to postcondition `Q`. Specifically, it says:

- From the precondition `P` we can obtain α `x` for some `x`, by opening the invariants `Eo \ Ei`.
- Having done so, it is possible to restore `P` by reestablishing α `x` and closing the invariants.
- Alternatively, by instead establishing β `x y` for any `y`, we my establish `Q x y` by closing the invariants.

The idea is that if an operation `e` performs a logically-atomic update from α to β, then for any given `P` and `Q` such that `atomic_shift α β Ei Eo P Q` we have $\{\mathsf{P}\}\ \mathsf{e}\ \{\mathsf{Q}\}$. Such

[1] For exposition, we elide some parameters of the predicates.

an operation thus consists of steps that may access α x but must preserve it, followed by a step that updates α x to β x y, followed by steps that cannot violate the (arbitrary) postcondition Q. This idea is expressed in the definition of logically-atomic weakest precondition:

```
Definition atomic_wp {A : Type}
  (α: A → iProp Σ)          (* atomic pre-cond. *)
  (β: A → val _ → iProp Σ) (* atomic post-cond. *)
  (Ei Eo : coPset)          (* in/out invs *)
  (e: expr _) : iProp Σ :=
  (∀ P Q, atomic_shift α β Ei Eo P Q -∗
    P -∗ WP e {{ v, ∃ x: A, Q x v }}).
```

4.4.2 Logically-atomic Specifications for the Dartino Queue Operations

Using this definition of logical atomicity, we can finally show the following specification for enqueue:

```
Lemma enqueue_spec :
  ∀ q p γ γ' E, procs_inv N γp ∗ qproc N γp γq p ⊢
  atomic_wp
     (λ l => ▷ queue N γp γq q γ γ' l)
     (λ l ret => queue N γp γq q γ γ' (l++[p]) ∗
                 ret = is_nil l)
     ∅ E
     enqueue (qhead q) (qtail q) (qsent q)
             (procAddrT_to_val p).
```

This specification establishes that enqueue atomically adds the process p to the end of the queue q, with the return value indicating whether the queue was empty at the time. However, the qproc predicate does not form part of the atomic precondition. Instead, it is in the overall precondition. This means that ownership of the qproc predicate is transferred to enqueue when it is called, rather than at the point it performs the atomic update. (This is analogous to the generalization of atomic triples in [2] that permits this kind of resource transfer.) The implementation can thus use these particular resources as it sees fit, without being concerned with interference from other threads. The overall precondition also establishes the invariant procs_inv.

Note that the atomic precondition is guarded under the ▷ modality. Since we have $P \vdash \triangleright P$, we could derive a specification without ▷. However, in Iris when an invariant is opened with a view shift, the contents is guarded by the ▷ modality. Therefore it is more convenient for clients that the atomic precondition should be guarded by ▷.

We can also show the following specification for dequeue:

```
Lemma dequeue_spec :
  ∀ q γ γ' E, procs_inv N γp
  ⊢ atomic_wp
    (λ l => ▷ queue N γp γq q γ γ' l)
    (λ l ret =>
      ∃ p l', l = p :: l' ∗ queue N γp γq q γ γ' l' ∗
        qproc N γp γq p ∗ ret = Some p
      ∨ l = [] ∗ queue N γp γq q γ γ' [] ∗ ret = NONE)
    ∅ E dequeue (qhead q) (qtail q) (qsent q) ().
```

Note that the atomic postcondition consists of a disjunction of the two cases: either the queue was non-empty and the process at the head of the queue is dequeued and returned; or the queue was empty, it is unchanged and the value NONE is returned.

Finally, we present the specification for tryDequeueEntry:

```
Lemma tryDequeueEntry_spec :
  ∀ q p γ γ' E, procs_inv N γp ∗ proc_inv N γp γq p
  ⊢ atomic_wp
      (λ l => ▷ queue N γp γq q γ γ' l)
      (λ l ret => (p ∈ l ∗ ∃ l1 l2, l = l1 ++ p :: l2 ∗
                  queue N γp γq q γ γ' (l1 ++ l2) ∗
                  qproc N γp γq p ∗ ret = true) ∨
                  (p ∉ l ∗ queue N γp γq q γ γ' l ∗
                  ret = false))
      (n_inv_proc N p) E
      tryDequeueEntry (qhead q) (qtail q) (qsent q)
                      (procAddrT_to_val p).
```

As with dequeue, the atomic postcondition is a disjunction: if the process p is in l, then the list l can be split such that l = l1++p :: l2, so we update the queue to l1++l2, extract the qproc resource for p, and return true; if p is not in l, we do nothing and return false.

Note that the global precondition requires the proc_inv invariant for the process we wish to dequeue, in addition to the procs_inv invariant present in the other specifications. This is since otherwise we would have no guarantee that p indeed represents a legitimate process object.

Interestingly, we also require that the invariant for the process is closed when obtaining the atomic pre-condition. This is because we have to case on the process being in the queue, which requires us to open the invariant. Since it is unsound to open the invariant twice, we need to enforce that the client does not open the invariant for the process.

5 Client

Logically atomic specifications allow clients to build and enforce their own protocol on top of data-structures. To illustrate this, we will consider a simple client of the Dartino Queue that simulates a round-robin scheduler. To simulate executing a process, we define a function doWork that simply reads and writes a process's pval pointer. We also define a function scheduler, which loops attempting to dequeue, "execute" and re-enqueue a process from a given queue, and a function enqueuer, which loops creating fresh processes and enqueuing them.

```
Definition doWork : val :=
  λ: pval,
  pval <- !pval.

Definition scheduler : val :=
  rec: loop h t s :=
    let: p := dequeue h t s () in
    match: p with
      NONE => ()
    | SOME p' => p' ;;
```

```
      doWork (pval p') ;;
      enqueue h t s p'
    end ;;
    loop h t s.

Definition enqueuer : val :=
  rec: loop h t s :=
    let: p := makeProc 1 in
    enqueue h t s p ;;
     loop h t s.

Definition concurrent_client : val :=
  λ: <>,
     let: q1 := makeQueue () in
     let: q1h := queue_head q1 in
     let: q1t := queue_tail q1 in
     let: q1s := queue_sent q1 in
     Fork (scheduler q1h q1t q1s) ;;
     Fork (scheduler q1h q1t q1s) ;;
     Fork (enqueuer q1h q1t q1s) ;;
```

The `concurrent_client` function creates a new Dartino Queue and forks two `scheduler` threads to run processes from the queue, and one `enqueuer` thread to add processes to the queue. (Recall that `queue_head`, `queue_tail` and `queue_sent` are projection functions from the tuple that represents a queue address.)

For `doWork`, to read and write a process's `pval` member, our custom protocol needs to transfer ownership of the reference to `doWork`. Similarly, for `enqueuer` and `scheduler` to operate on the same queue, the protocol should allow for each to access the queue, to transfer ownership of the process's `pval` to the shared state when enqueueing a process, and to remove the ownership of `pval` when dequeuing a process.

The following invariant `client_queue_inv` is an excellent candidate for the shared state for our custom protocol:

```
Definition client_queue γp γq q γ γ' l :=
  queue N γp γq q γ γ' l ∗
        [∗ list] p ∈ l, ∃ v, pvall p ↦{1/2} v

Definition client_queue_inv γp γq (q : queueAddrT) γ γ' :=
  inv (n_inv_queue q) (∃ l, client_queue γp γq q γ γ' l).
```

This invariant holds a ½ fraction of the `pval` pointer for each process in the queue. (The remaining ½ belongs to the `procs_inv` invariant, and can be obtained from the `qproc` resource when a process is removed from the queue.)

To show how this custom protocol works, consider how the `scheduler` function will use the logically atomic specification for `dequeue`. To do so, it must establish an `atomic_shift` α β `Ei Eo P Q`, where α, β and `Ei` are determined by the `dequeue` specification as:

```
α := (λ l => ▷ queue N γp γq q γ γ' l)
β := (λ l ret =>
    ∃ p l', l = p :: l' ∗ queue N γp γq q γ γ' l' ∗
      qproc N γp γq p ∗ ret = Some p
    ∨ l = [] ∗ queue N γp γq q γ γ' [] ∗ ret = NONE)
Ei := ∅
```

and `Eo`, `P` and `Q` are determined by the client as:

```
Eo := { n_inv_queue q }
P := True
Q := λ l ret => l = [] ∗ ret = NONE ∨
             ∃ p l' v, l = p :: l' ∗ qproc N γp γq p ∗
                    ▷ (pvall p) ↦{1/2} v ∗ ret = Some p
```

The precondition is `True` since the queue belongs to the client invariant, which is persistent. The postcondition extracts the `qproc` and `pval` pointer resources from the queue when the operation succeeds. The client obtains `P` and `Q` as a pre- and postcondition for dequeue by establishing the atomic shift, namely:

```
(P ={Eo, Ei}=> ∃ x:A, α x ∗
  ((α x ={Ei, Eo}=∗ P) ∧
    (∀ y, β x y ={Ei, Eo}=∗ Q x y)))
```

Since `Eo` is `{ n_inv_queue q }` and `Ei` is ∅, the view shift opens the client invariant to obtain α. Recall that opening an invariant obtains its resources guarded under the later modality (▷), and hence its presence in α. The wand shifts close the client invariant, the first when no update is performed, and the second when the dequeue operation takes effect. For the latter, when the operation succeeds the dequeued process is no longer in the queue, and we have

```
queue N γp γq q γ γ' l ∗
▷ [∗ list] p' ∈ p::l, ∃ v, pvall p' ↦{1/2} v
```

To close the invariant again, we unfold the iterated separating conjunction (`[∗ list]`) to extract the `pval` resource for the process `p` that is being dequeued:

```
queue N γp γq q γ γ' l ∗ ∃ v', ▷ pvall p' ↦{1/2} v' ∗
▷ [∗ list] p' ∈ l, ∃ v, pvall p' ↦{1/2} v
```

The client invariant can then be closed and the

```
▷ pvall p' ↦{1/2} v'
```

resource can be retained by the postcondition `Q`.

6 Conclusion

We have formally specified and verified the concurrent queue data structure at the heart of the Dartino Framework using Iris in the Coq proof assistant. While the algorithm itself is fairly simple, giving a reasonable specification for it is not trivial. For this, we have used an encoding of logical atomicity in Iris. Logical atomicity allows us to precisely capture the behaviour of the queue operations, allowing clients of the data structure to impose their own invariants. We demonstrate this by verifying a concurrent client using our specification. Our work is a case study which shows that Iris and logical atomicity can be effectively applied to reason about real-world code.

References

[1] The Coq Development Team. 2016. *The Coq Proof Assistant Reference Manual*. http://coq.inria.fr Version 8.6.

[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.

[3] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492.

[4] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 637–650.

[5] Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The essence of higher-order concurrent separation logic. In *European Symposium on Programming*. Springer, 696–723.

[6] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *ACM SIGPLAN-SIGACT Symposium om Principles of Programming Languges (POPL 2017)*. ACM.

[7] Peter O'Hearn, John Reynolds, and Hongseok Yang. 2001. Local reasoning about programs that alter data structures. In *Computer science logic*. Springer, 1–19.

[8] John C Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*. IEEE, 55–74.