TITLE HERE

Mathias Pedersen, 201808137

Master's Thesis, Computer Science
May 2024
Advisor: Amin Timany

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

►in English. . . ◄

Resumé

▶in Danish. . . ◀

Acknowledgments

►. . . ◄

Mathias Pedersen
Aarhus, May 2024.

Contents

Chapter 1

Introduction

►motivate and explain the problem to be addressed◄

►example of a citation: [1]◄ ►get your bibtex entries from `https://dblp.org/`◄

Chapter 2

Preliminaries

▶Mention that the project uses heaplang and the program logic iris, and hence we need to know about them◀

2.1 HeapLang

▶Write about heaplang◀ ▶Talk about Syntactic sugar: i.e. e1 ;; e2 = (lam v, e2) e1 where v is fresh, and CAS ... as Snd (CMPXHG ...), and derived rules for them.◀

▶Question: should formal definition of heaplang be in section, appendix, or reference to ILN?◀

2.2 The Iris Program Logic Framework

▶Write about Iris◀ ▶Seperation logic◀ ▶Present some of the derivation rules◀ ▶Present hoare Triples and Weakest Pre-condition◀ ▶Persistent and Later◀ ▶For Later, explain that it ties propositions to program-steps. Explain the löb induction rule, mention the intuition when P is hoare triple◀ ▶Resource Algebra◀ ▶Invariants◀ ▶Fancy update modality and viewshift◀

2.3 Formalisation in Coq

▶Mention that Iris is formalised in Coq◀ ▶Stuff works in terms of weakest precondition◀ ▶Mention that all the work done in this project has also been mechanised in Coq◀ ▶give an overview of how the coq files relate to each chapter/section◀

Chapter 3

The Two-Lock Michael Scott Queue

I present here he an implementation of the Two-lock MS-Queue in HeapLang. This implementation differs slightly from the original, presented in [1], but most changes simply reflect the differences in the two languages.

3.1 Preliminaries

The underlying data structure making up the queue is a singly-linked list. The linked-list will always contain at least one element, called the *sentinel* node, marking the beginning of the queue. Note that the sentinel node is itself not part of the queue, but all nodes following it are. The queue keeps a head pointer (ℓ_{head}) which always points to the sentinel, and a tail pointer (ℓ_{tail}) which points to some node in the linked list.

In my implementation, a node can be thought of as a triple $(\ell_{i\_\text{in}}, v_i, \ell_{i\_\text{out}})$. The location ℓ_{i_in} points to the pair $(v_i, \ell_{i\_\text{out}})$, where v_i is the value of the node, and ℓ_{i_out} either points to None which represents the null pointer, or to the next node in the linked list. When we say that a location ℓ points to a node $(\ell_{i\_\text{in}}, v_i, \ell_{i\_\text{out}})$, we mean that $\ell \mapsto \ell_{i\_\text{in}}$. Hence, if we have two adjacent nodes $(\ell_{i\_\text{in}}, v_i, \ell_{i\_\text{out}})$, $(\ell_{i+1\_\text{in}}, v_{i+1}, \ell_{i+1\_\text{out}})$ in the linked list, then we have the following structure: $\ell_{i\_\text{in}} \mapsto (v_i, \ell_{i\_\text{out}})$, $\ell_{i\_\text{out}} \mapsto \ell_{i+1\_\text{in}}$, and $\ell_{i+1\_\text{in}} \mapsto v_{i+1}, \ell_{i+1\_\text{out}}$.

The reader may wonder why there is an extra, intermediary "in" pointer, between the pairs of the linked list, and why the "out" pointer couldn't point directly to the next pair. In the original implementation [1], nodes are allocated on the heap. To simulate this in HeapLang, when creating a new node, we create a pointer to a pair making up the node. Now, in the C-like language used in the original specification, an assignment operator is available which is not present in HeapLang. So in order to mimic this behaviour, we model variables as pointers. In this way, we can model a variable x as a location ℓ_x, and the value stored at ℓ_x is the current value of x. This means that the variable ℓ_{i_out} (called "next" in the original) becomes a location ℓ_{head}, and the value stored at the location is what head is currently assigned to. Since ℓ_{i_out} is supposed to be a variable containing a pointer, then the value saved at that location will also be a pointer.

3.2 Implementation

The queue consists of 3 functions: initialize, enqueue, and dequeue, and as the name of the data structure suggests, the functions rely on two locks. To this end, we shall assume that we have some lock implementation given. In the accompanying coq mechanisation, a "spin-lock" is used, but the only part we really care about is its specification; this can be found in Example 8.38 in **►Cite Iris Lecture Notes◄**.

3.2.1 initialize

initialize will first create a single node – the sentinel – marking the start of the linked list. It then creates two locks, H_lock and T_lock, protecting the head and tail pointers, respectively. Finally, it creates the head and tail pointers, both pointing to the sentinel. The queue is then a pointer to a structure containing the head, the tail, and the two locks.
Figure 3.1 illustrates the structure of the queue after initialisation. Note that one of the pointers is coloured blue. This represents a *persistent* pointer; a pointer that will never be updated again. All "in" pointers ℓ_{i_in}, are persistent, meaning that, once created, they will only ever point to $(v_i, \ell_{i\_\text{out}})$. We shall use the notation $\ell \mapsto^{\square} v$ (introduced in [2]) to mean that ℓ points persistently to v.

Note that in the original specification, a queue is a pointer to a 4-tuple $(\ell_{head}, \ell_{tail}, H\_lock, T\_lock)$. Since HeapLang doesn't support 4-tuples, we instead represent the queue as a pointer to a pair of pairs: $((\ell_{head}, \ell_{tail}), (H\_lock, T\_lock))$.

3.2.2 enqueue

To enqueue a value, we must create a new node, append it to the underlying linked-list, and swing the tail pointer to this new node. These three operations are depicted in figure 3.2.

enqueue takes as argument the value to be enqueued and creates a new node containing this value (corresponding to figure 3.2a). This creation doesn't interact with the underlying queue data-structure, hence why we don't acquire the T_lock first. After creating the new node, we must make the last node in the linked list point to it. Since this operation interacts with the queue, we first acquire the T_lock. Once we obtain the lock, we make the last node in the linked list point to our new node (figure 3.2b). Following this, we swing ℓ_{tail} to the new last node in the linked list (figure 3.2c).

Figure 3.2 also illustrates when pointers become persistent; once the previous last node is updated to point to the newly inserted node, that pointer will never be updated again, hence becoming persistent.

3.2.3 dequeue

It is of course only possible to dequeue an element from the queue if the queue contains at least one element. Hence, the first thing dequeue does is check if the queue is empty. We can detect an empty queue by checking if the sentinel is the last node in the linked list. Being the last node in the linked list corresponds to having the "out" node be None. If this is the case, then the queue is empty and the code returns None. Otherwise, there is a node just after the sentinel, which is the first node of the queue. To dequeue it, we first read the associated value, and next we swing the head to it, making it the new sentinel. Finally, we return the value we read.

Since all of these operations interact with the queue, we shall only perform them after having acquired H_lock.

Figure 3.3 illustrates running dequeue on a non-empty queue. Note that the only change is that the head pointer is swung to the next node in the linked list; the old sentinel is not deleted, it just become unreachable from the heap pointer. In this way, the linked list only ever grows.

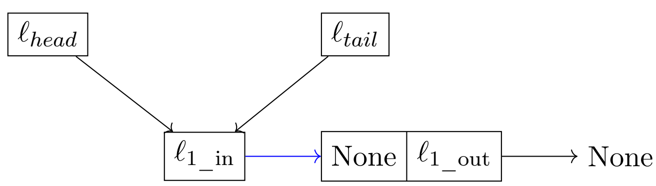
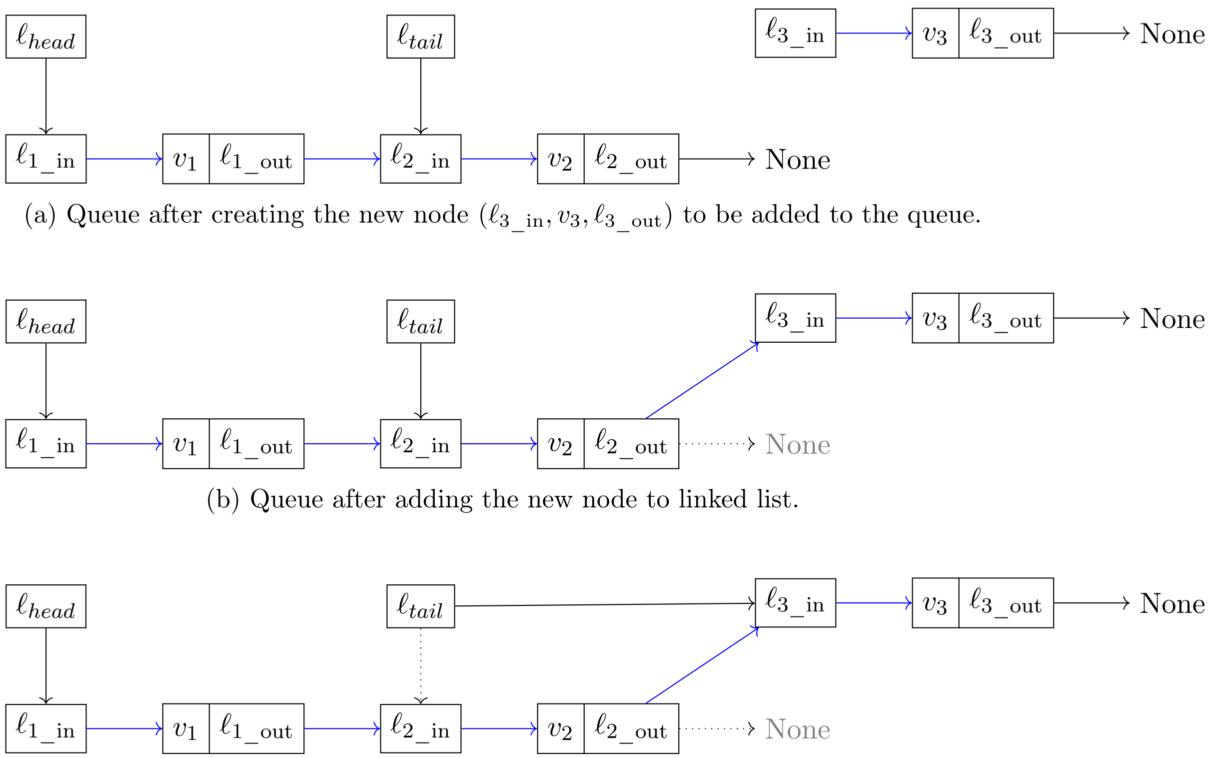


Figure 3.1: Queue after initialisation

```
initialize ≜
  let node = ref (None, ref (None)) in
  let H_lock = newlock() in
  let T_lock = newlock() in
  ref ((ref (node), ref (node)), (H_lock, T_lock))
```

```
enqueue Q value ≜
  let node = ref (Some value, ref (None)) in
  acquire(snd(snd(! Q)));
  snd(!(!(snd(fst(! Q))))) ← node;
  snd(fst(! Q)) ← node;
  release(snd(snd(! Q)))
```

```
dequeue Q ≜
  acquire(fst(snd(! Q)));
  let node = !(fst(fst(! Q))) in
  let new_head = !(snd(! node)) in
  if new_head = None then
    release(fst(snd(! Q)));
    None
  else
    let value = fst(! new_head) in
```



(a) Queue after creating the new node $(\ell_{3\_in}, v_3, \ell_{3\_out})$ to be added to the queue.

(b) Queue after adding the new node to linked list.

(c) Queue after swinging tail pointer to the new node.

Figure 3.2: Enqueuing an element to a queue with one element.

$\quad\quad \mathsf{fst}(\mathsf{fst}(!\,Q)) \leftarrow \mathit{new\_head};$
$\quad\quad \text{release}(\mathsf{fst}(\mathsf{snd}(!\,Q)));$
$\quad\quad \mathit{value}$

3.3 Sequential Specification

Let us first prove a specification for the two-lock michael scott queue in the simple case where we don't allows for concurrency. In this case, we know that only a single thread will interact with the queue at any given point in a sequential manner. This means that we give a specification that tracks the exact contents of the queue. To this end, we shall define the abstract state of the queue, denoted xs_v as a list of HeapLang values. I.e. xs_v : *List Val*. We adopt the convention that enqueueing an element is done by adding it to the front of the list, and dequeueing removes the last element of the list (if such an element exists). The reason for this choice is purely technical.

Since the queue uses two locks, we will get two ghost names; one for each lock. For this specification, these are the only two ghost names we will need. However, for the later specifications, we will use more resource algebra, and will need more ghost names. Thus, to ease notation, we shall define the type "*SeqQgnames*" whose purpose is to keep track of the ghost names used for a specific queue. Since we only have two ghost names for this specification, elements of *SeqQgnames*

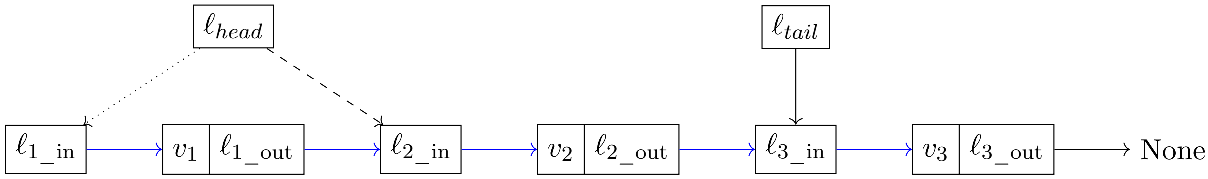


Figure 3.3: Dequeueing an element (v_2) from a queue with two elements (v_2, v_3). The dotted line represents the state before the dequeue, and the dashed line is the state after dequeuing.

will simply be pairs. For an element $Q_\gamma \in \mathit{SeqQgnames}$, the first element of the pair, written $Q_\gamma.\gamma_{Hlock}$, will contain the ghost name for the head lock, and the second element, $Q_\gamma.\gamma_{Tlock}$, the ghost name for the tail lock.

The sequential specification we wish to prove is the following:

Lemma 1 (Two-Lock M&S-Queue Sequential Specification)**.**

$$
\begin{aligned}
&\exists \text{is\_queue\_seq} : \mathit{Val} \to \mathit{List\ Val} \to \mathit{SeqQgnames} \to \mathsf{Prop}.\\
&\quad \{\mathsf{True}\}\ \text{initialize}()\ \{v_q. \exists Q_\gamma.\ \text{is\_queue\_seq}\ \ v_q\ []\ Q_\gamma\}\\
\wedge &\quad \forall v_q, v, xs_v, Q_\gamma.\ \{\text{is\_queue\_seq}\ \ v_q\ xs_v\ Q_\gamma\}\ \text{enqueue}\ \ v_q\ v\ \{w.\ \text{is\_queue\_seq}\ \ v_q\ (v :: xs_v)\ Q_\gamma\}\\
\wedge &\quad \forall v_q, xs_v, Q_\gamma.\ \{\text{is\_queue\_seq}\ \ v_q\ xs_v\ Q_\gamma\}\\
&\qquad\qquad\qquad \text{dequeue}\ \ v_q\\
&\qquad\qquad\qquad \left\{ v.\ \begin{array}{l} (xs_v = [] * v = \text{None} * \text{is\_queue\_seq}\ \ v_q\ xs_v\ Q_\gamma) \vee \\ (\exists x_v, xs'_v.\ xs_v = xs'_v \mathbin{++} [x_v] * v = \text{Some}\, x_v * \text{is\_queue\_seq}\ \ v_q\ xs'_v\ Q_\gamma) \end{array} \right\}
\end{aligned}
$$

The predicate is_queue_seq v_q xs_v Q_γ captures that the value v_q is a queue, whose content matches that of our abstract representation xs_v, and the queue uses the ghost names described by Q_γ. Note that the is_queue_seq predicate is not required to be persistent, hence it cannot be duplicated and given to multiple threads. This is the sense in which this specification is sequential.

3.4 Proving the Sequential Specification

3.4.1 The is_queue_seq Predicate

To prove the specification we must give a specific is_queue_seq predicate. To help guide us in designing this, we give the following observations about the behaviour of the implementation.

1. Head always points to the first node in the queue.
2. Tail always points to either the last or second last node in the queue.
3. All but the last pointer in the queue (the pointer to None) never change.

Observation 2 captures the fact that, while enqueueing, a new node is first added to the linked list, and then later the tail is updated to point to the newly added node. Since only one thread can enqueue a node at a time (due to the lock), then the tail will only ever point to the last or second last due to the above. However, in a sequential setting, the tail will always appear to point to the last node, as no one can inspect the queue while the tail points to the second last.

Insight 3 means that we can mark all pointers in the queue (except the pointer to the null node) as persistent. This is technically not needed in the sequential case, but we will incorporate it now, as we will need it in the concurrent setting.

Definition 3.4.1 (Two-Lock M&S-Queue - is_queue_seq Predicate)**.**

$$
\begin{aligned}
\text{is\_queue\_seq } v_q\ xs_v\ Q_\gamma \triangleq\ & \exists \ell_{queue}, \ell_{head}, \ell_{tail} \in \mathit{Loc}.\ \exists h_{lock}, t_{lock} \in \mathit{Val}. \\
& v_q = \ell_{queue} * \ell_{queue} \mapsto^{\Box} ((\ell_{head}, \ell_{tail}), (h_{lock}, t_{lock})) * \\
& \exists xs_{queue} \in \mathit{List}(\mathit{Loc} \times \mathit{Val} \times \mathit{Loc}).\ \exists x_{head}, x_{tail} \in (\mathit{Loc} \times \mathit{Val} \times \mathit{Loc}). \\
& \mathit{proj\_val}\ xs_{queue} = \mathit{wrap\_some}\ xs_v * \\
& \text{isLL}(xs_{queue} +\!\!+ [x_{head}]) * \\
& \ell_{head} \mapsto (\text{in } x_{head}) * \\
& \ell_{tail} \mapsto (\text{in } x_{tail}) * \mathit{isLast}\ x_{tail}\ (xs_{queue} +\!\!+ [x_{head}]) * \\
& \mathit{isLock}\ Q_\gamma.\gamma_{Hlock}\ h_{lock}\ \mathsf{True} * \\
& \mathit{isLock}\ Q_\gamma.\gamma_{Tlock}\ t_{lock}\ \mathsf{True}.
\end{aligned}
$$

This is_queue_seq predicate states that the value v_q is a location, which persistently points to the structure containing the head, the tail, and the two locks. It also connects the abstract state xs_v with the concrete state by stating that if you strip away the locations in xs_{queue} (achieved by $\mathit{proj\_val}$) and wrap the values in the abstract state xs_v in Some (achieved by $\mathit{wrap\_some}$), then the lists become equal.
Next, the predicate specifies the concrete state. There is some head node x_{head}, which the head points to. This head node and the nodes in xs_{queue} form the underlying linked list (specified using the isLL predicate below). There is also a tail node, which is the last node in the linked list, and the tail points to this node. The proposition $\mathit{isLast}\ x\ xs$ simply asserts the existence of some xs', so that $xs = x :: xs'$.
Finally, we have the isLock predicate for our two locks. Since we are in a sequential setting, then the locks are superfluous, hence they simply protect True.

The isLL predicate essentially creates the structure seen in the examples of section 3.2. It is defined in two steps. Firstly, we create all the persistent pointers in the linked list using the isLL_chain predicate. Note that this in effect makes isLL_chain xs persistent for all xs.

Definition 3.4.2 (Linked List Chain Predicate)**.**

$$
\begin{aligned}
\text{isLL\_chain } [] &\equiv \mathsf{True} \\
\text{isLL\_chain } [x] &\equiv \text{in } x \mapsto^{\Box} (\text{val } x, \text{out } x) \\
\text{isLL\_chain } x :: x' :: xs &\equiv \text{in } x \mapsto^{\Box} (\text{val } x, \text{out } x) * \text{out } x' \mapsto^{\Box} \text{in } x * \text{isLL\_chain } x' :: xs
\end{aligned}
$$

Then, to define isLL, we add that the last node in the linked list points to None.

Definition 3.4.3 (Linked List Predicate)**.**

$$\begin{aligned} \text{isLL}\,[] &\equiv \mathit{True} \\ \text{isLL}\, x :: xs &\equiv \text{out}\ \ x \mapsto \text{None} * \text{isLL\_chain}\ \ x :: xs \end{aligned}$$

For instance, if we wanted to capture the linked list in figure 3.2c, we would use the list $xs = [(\ell_{3\_\text{in}}, v_3, \ell_{3\_\text{out}}); (\ell_{2\_\text{in}}, v_2, \ell_{2\_\text{out}}); (\ell_{1\_\text{in}}, v_1, \ell_{1\_\text{out}})]$. isLL xs will expand to $\ell_{3\_\text{out}} \mapsto \text{None} * \text{isLL\_chain}\ xs$, and isLL_chain xs expands to

$$\begin{aligned} &\ell_{3\_\text{in}} \mapsto^{\Box} (x_3, \ell_{3\_\text{out}}) * \ell_{2\_\text{out}} \mapsto^{\Box} \ell_{3\_\text{in}} * \\ &\ell_{2\_\text{in}} \mapsto^{\Box} (x_2, \ell_{2\_\text{out}}) * \ell_{1\_\text{out}} \mapsto^{\Box} \ell_{2\_\text{in}} * \\ &\ell_{1\_\text{in}} \mapsto^{\Box} (x_1, \ell_{1\_\text{out}}) \end{aligned}$$

Note how this matches the structure of the linked list in figure 3.2c.

The proofs require us manipulate specific isLL predicates quite a bit – appendix **▶Add appendix and refer to it◀** shows the specific lemmas we will use, but the proof outlines will generally not mention the lemmas explicitly.

3.4.2 Proof outline

Initialise

Proving the initialise spec amounts to stepping through the code, giving us the required resources, and then using these to create an instance of is_queue_seq with the obtained resources. To begin with, we step through the lines creating the first node x_1, giving us locations ℓ_{1_in}, ℓ_{1_out} with $\ell_{1\_\text{out}} \mapsto \text{None}$ and $\ell_{1\_\text{in}} \mapsto (\text{None}, \ell_{1\_\text{out}})$. We can then update the latter points-to predicate to become persistent, giving us $\ell_{1\_\text{in}} \mapsto^{\Box} (\text{None}, \ell_{1\_\text{out}})$. We then step to the creation of the two locks, where we shall use the newlock specification asserting that the locks should protect True. This gives us two ghost names, γ_{Tlock}, γ_{Tlock}, which we will collect in a *SeqQgnames* pair, Q_γ. Next, we step through the allocations of the head, tail, and queue, which gives us locations ℓ_{head}, ℓ_{tail}, ℓ_{queue}, such that both ℓ_{head} and ℓ_{tail} point to node x_1, and such that ℓ_{queue} points to the structure containing the head, tail, and two locks. This last points to predicate we update to become persistent. With this, we now have all the resources needed to prove the post-condition: $\exists Q_\gamma.\ \text{is\_queue\_seq}\ \ell_{queue}\ Q_\gamma$. Proving this follows by a sequence of framing away the resources we obtained and instantiating existentials with the values we got above. Most noteworthy, we pick the empty list for xs_{queue}, and node x_1 for x_{head} and x_{tail}.

Enqueue

▶add line numbers to code, and refer to them in proof◀ For enqueue, we get in our pre-condition is_queue_seq $v_q\ xs_v\ Q_\gamma$, and we wish to that, if we run enqueue $v_q\ v$, then we will get is_queue_seq $v_q\ (v :: xs_v)\ Q_\gamma$. The proposition is_queue_seq $v_q\ xs_v\ Q_\gamma$ gives us all the resources we will need to step through the code. Firstly, we create a new node, node x_{new}, with val $x_{new} = v$. We then have to acquire the lock, which will just give us True.

The next line adds node x_new to the linked list, by first finding the tail, from the queue pointer ℓ_{queue}, and then finding the node that the tail points to, denoted x_{tail}, and finally writing updating the out location of x_{tail} to point to x_new. The resources needed to do this are all described in is_queue_seq v_q xs_v Q_γ. Firstly, it tells us that ℓ_{queue} points to the structure containing ℓ_{tail}. Secondly, it tells us that ℓ_{tail} points to x_{tail}, which is the last node in the linked list $(xs_{queue} \mathbin{++} [x_{head}])$. Thirdly, since we know that x_{tail} is the last node in the linked list, then by the isLL predicate, we know that x_{tail} points to None and that it has the node-like structure described by isLL_chain . This is all we need to step through the line, adding x_{new} to the linked list. After performing the write, we then get that x_{tail} points to x_{new}, instead of None. We make this points-to predicate persistent.

The next line swings the tail to x_{new}. As describe above, we already know that ℓ_{tail} points to x_{tail}, so we have the required resources to perform the write. Afterwards, we get that ℓ_{tail} points to x_{new}.

Finally, we release the lock using the release specification (and we simply give back True), and the only thing left is to prove the postcondition: is_queue_seq v_q $(v :: xs_v)$ Q_γ. For the existentials, we shall pick the ones we got from the precondition, with the exception for xs_{queue} and x_{tail}. For xs_{queue}, we shall use the same xs_{queue} we got from the precondition, but with xs_new cons'ed to it, and for x_{tail}, we chose the new tail node: x_{new}. With these choices, proving is_queue_seq v_q $(v :: xs_v)$ Q_γ is fairly straightforward.

Dequeue

For dequeue v_q, our precondition is is_queue_seq v_q xs_v Q_γ, and our post condition states that either the queue is empty, or there is a tail element which is returned by the function, and removed from the queue.

Stepping through the function, we first do the superfluous acquire. Next, we get the head node x_{head} through the queue pointer ℓ_{queue}. As described above for Enqueue, we get the resource to do this through is_queue_seq v_q xs_v Q_γ. The is_queue_seq predicate also tells us that x_{head} is a node in the linked list (described by the isLL predicate), hence we can step through the code in the next line, which finds the node that x_{head} is pointing to. Now, depending on whether or not the queue is empty, x_{head} either points to None, or some node x_{head_next}. Thus, we shall perform a case analysis on xs_{queue}.

xs_{queue} **is empty**: In this case, we will have that isLL$[x_{head}]$, which tells us that x_head points to None. Hence, the "then" branch of the "if" will be taken. This branch simply releases the lock and returns None. In this case, we prove the first disjunction in the post-condition. Since xs_v is reflected in xs_{queue}, then we will be able to conclude that xs_v is empty, and since we haven't modified the queue, we can create is_queue_seq v_q xs_v Q_γ using the same resources we got from the pre-condition.

xs_{queue} **is not empty**: In this case, we can conclude that there must be some node x_{head_next}, which is the first node in xs_{queue}. I.e. $xs_{queue} = xs'_{queue} \mathbin{++} [x_{head\_next}]$. We can thus use the isLL predicate to conclude that x_{head} must point to x_{head_next}. Hence the else branch will be taken. Since

x_{head_next} is part of the linked list, then isLL tells us it has the node-like structure, allowing us to extract its value in the first line of the else branch.
In the next line, we make the head pointer, ℓ_{head} point to x_{head_next}, and we have the resource to do this through is_queue_seq v_q xs_v Q_γ.
Finally, we release the lock and return the value we got from x_{head_next}. We must now prove the post-condition, and this time we prove the second disjunct. Since xs_v is reflected in xs_{queue}, then it must also be the case that xs_v is non-empty, and it has a first element, x_v, which is related to the first element of xs_{queue}, i.e. x_{head_next}. This allows us to conclude that the returned value (val x_{head_next}) is exactly x_v, but wrapped in a Some, as we had to prove. Finally, we must prove is_queue_seq v_q xs'_v Q_γ, where xs'_v is xs_v but with x_v removed. For the existentials, we pick the same values we got from the precondition, wit the exception of xs_{queue} and x_{head}. For xs_{queue} we pick the same xs_{queue} we got from the precondition, but with the first element, x_{head_next} removed. By doing this, xs_{queue} will be reflexed in xs'_v. For x_head, we pick the new head, which we have obtained that ℓ_{head} points to: x_{head_next}. With these choices, we can prove the predicate.

3.5 Concurrent Specification

For the concurrent specification, we will need the predicate capturing the queue (here denoted is_queue_conc) to be duplicable. To achieve this, we shall initially give up on tracking the abstract state of the queue, and instead add a predicate Φ, which we will ensure holds for all elements of the queue. In this way, when dequeueing, we at least know that if we get some value, then Φ holds of this value. The specification we wish to prove is as follows.

Lemma 2 (Two-Lock M&S-Queue Concurrent Specification)**.**

$$
\begin{aligned}
&\exists\, \text{is\_queue\_conc} : (\mathit{Val} \to \textsf{\textit{Prop}}) \to \mathit{Val} \to \mathit{ConcQgnames} \to \textsf{\textit{Prop}}.\\
&\forall \Phi : \mathit{Val} \to \textsf{\textit{Prop}}.\\
&\qquad \forall v_q, Q_\gamma.\ \text{is\_queue\_conc}\ \ \Phi\ v_q\ Q_\gamma \implies \Box\, \text{is\_queue\_conc}\ \ \Phi\ v_q\ Q_\gamma\\
&\wedge \quad \{\textsf{\textit{True}}\}\ \text{initialize}()\ \{v_q.\exists Q_\gamma.\ \text{is\_queue\_conc}\ \ \Phi\ v_q\ Q_\gamma\}\\
&\wedge \quad \forall v_q, v, Q_\gamma.\ \{\text{is\_queue\_conc}\ \ \Phi\ v_q\ Q_\gamma * \Phi\ v\}\ \text{enqueue}\ \ v_q\ v\ \{v.\textsf{\textit{True}}\}\\
&\wedge \quad \forall v_q, Q_\gamma.\ \{\text{is\_queue\_conc}\ \ \Phi\ v_q\ Q_\gamma\}\ \text{dequeue}\ \ v_q\ \{v.v = \text{None} \vee (\exists x_v.\ v = \text{Some}\, x_v * \Phi\ x_v)\}
\end{aligned}
$$

Note that the type of the collection of ghost names here is $ConcQgnames$, as we will require more ghost names than before. The new ghost names are used for "tokens" which are introduced in the following section.

3.6 Proving the Concurrent Specification

3.6.1 The is_queue_conc Predicate

As we did for the sequential specification, we note here some useful observations about the implementation.

1. Nodes in the linked list are never deleted. Hence, the linked list only ever grows.

2. The tail can lag one node behind Head.

3. At any given time, the queue is in one of four states:

    (a) No threads are interacting with the queue (**Static**)

    (b) A thread is enqueueing (**Enqueue**)

    (c) A thread is dequeuing (**Dequeue**)

    (d) A thread is enqueueing and a thread is dequeuing (**Both**)

Observation 2 might seem a little surprising, and indeed it stands in contrast to property 5 in [1], which states that the tail never lags behind head. I also didn't realise this possibility until a proof attempt using a model that "forgot" old nodes lead to an unprovable case (see section 3.6.2). The situation can occur when the queue is empty, and a thread performs an incomplete enqueue; it attaches the new node to the end, but before it can swing the tail to this new node, another thread performs a dequeue, which dequeues this new node, swinging the head to it. Now the tail is lagging a node behind the head.

It is not possible for the tail to point more than one node behind the head, as in order for this to happen, more nodes must be enqueued, but this can't happen before the current enqueue finishes, which will update the tail and bring it up to speed with the head.

Fortunately, this isn't an issue for safety, but a consequence of this possibility is that when modelling the queue, we must remember at least one "old" node (i.e. a dequeued node), as the tail might be pointing to this node. For the sake of simplicity in the model, the choice is made to remember an arbitrary amount of old nodes, which is represented by the list xs_{old}.

Observation 3 is a simple consequence of the implementation using two locks.

Since we want is_queue_conc to be persistent, then we cannot directly state the points-to predicates as we did in the sequential case. However, we will still need all the same resources to be able to prove the specification. The solution is to have an invariant which describes the concrete state of the queue. In the proofs, when we need access to some resource, we shall then access it by opening the invariant. We now present the invariant and explain it afterwards.

Definition 3.6.1 (Two-Lock M&S-Queue Concurrent Invariant).

$$
\begin{aligned}
&queue\_invariant\ \Phi\ \ell_{head}\ \ell_{tail}\ Q_\gamma \triangleq \\
&\exists xs_v.\ All\ xs_v\ \Phi * && \textit{(abstract state)} \\
&\exists xs, xs_{queue}, xs_{old}, x_{head}, x_{tail}. && \textit{(concrete state)} \\
&xs = xs_{queue} \mathbin{++} [x_{head}] \mathbin{++} xs_{old} * \\
&\text{isLL}\ \ xs * \\
&proj\_val\ xs_{queue} = wrap\_some\ xs_v * \\
&(\\
&\quad \ell_{head} \mapsto (\text{in}\ x_{head}) * \ell_{tail} \mapsto (\text{in}\ x_{tail}) * isLast\ x_{tail}\ xs * && \textbf{\textit{(Static)}} \\
&\quad \text{ToknE}\ \ Q_\gamma * \text{ToknD}\ \ Q_\gamma * \text{TokUpdated}\ \ Q_\gamma \\
&\ \vee \\
&\quad \ell_{head} \mapsto (\text{in}\ x_{head}) * \ell_{tail} \mapsto^{\frac{1}{2}} (\text{in}\ x_{tail}) * && \textbf{\textit{(Enqueue)}} \\
&\quad (isLast\ x_{tail}\ xs * \text{TokBefore}\ \ Q_\gamma \vee isSndLast\ x_{tail}\ xs * \text{TokBefore}\ \ Q_\gamma) * \\
&\quad \text{TokE}\ \ Q_\gamma * \text{ToknD}\ \ Q_\gamma \\
&\ \vee \\
&\quad \ell_{head} \mapsto^{\frac{1}{2}} (\text{in}\ x_{head}) * \ell_{tail} \mapsto (\text{in}\ x_{tail}) * isLast\ x_{tail}\ xs * && \textbf{\textit{(Dequeue)}} \\
&\quad \text{ToknE}\ \ Q_\gamma * \text{TokD}\ \ Q_\gamma * \text{TokUpdated}\ \ Q_\gamma \\
&\ \vee \\
&\quad \ell_{head} \mapsto^{\frac{1}{2}} (\text{in}\ x_{head}) * \ell_{tail} \mapsto^{\frac{1}{2}} (\text{in}\ x_{tail}) * && \textbf{\textit{(Both)}} \\
&\quad (isLast\ x_{tail}\ xs * \text{TokBefore}\ \ Q_\gamma \vee isSndLast x_{tail} xs * \text{TokBefore}\ \ Q_\gamma) * \\
&\quad \text{TokE}\ \ Q_\gamma * \text{TokD}\ \ Q_\gamma \\
&)
\end{aligned}
$$

In contrast to the sequential specification, the abstract state is now existentially quantified, hence the exact contents of the queue are not tracked. Instead, we have added the proposition $All\ xs_v\ \Phi$, which states that all values in xs_v (i.e. the values currently in the queue) satisfy the predicate Φ. This will allow us to conclude that dequeued values satisfy Φ.

The concrete state of the queue is still reflected in the abstract state through projecting out the values of the nodes ($proj\_val$), and wrapping the values in the queue in Some ($wrap\_some$). Another difference is that we now also keep track of an arbitrary number of "old" nodes; nodes that are behind the head node, x_{head}. As discussed above, this inclusion is due to observation 2.

As before, we also assert that the concrete state forms a linked list, as described by the isLL predicate.

The final part of the invariant describes the four possible states of the queue, as described in 3. Since the resources used by the queue are inside an invariant, and enqueueing/dequeueing threads need to access the resources of the queue multiple times, then we will have to open and close the invariant multiple times. Each time we open the invariant, the existentially quantified variables will not be the same as those from early accesses of the invariant (as they are existentially

quantified). Thus, the threads must be able to "match up" variables from previous accesses to later accesses. The way we shall achieve this is by allowing threads to keep a *fraction* of the points-to predicate that it is using. For instance, an enqueuing thread will have to access the points-to predicate concerning ℓ_{tail} multiple times, and in between accesses of the invariant, it can get to keep half of the points-to predicate. Thus, when it opens the invariant later, it will have $\ell_{tail} \mapsto^{\frac{1}{2}} \text{in } x_{tail}$ from an earlier access, and it will obtain the existence of some new x'_{tail}, such that $\ell_{tail} \mapsto^{\frac{1}{2}} \text{in } x'_{tail}$. Combining the two points-to predicates allows us to conclude that $\text{in } x_{tail} = \text{in } x'_{tail}$. In this way, we can match up variables from earlier accesses to variables in later accesses.
In the **Static** state where no thread is interacting with the queue, the queue owns all of the points-to predicates concerning the head and tail.
In the **Enqueue** state, the enqueueing thread owns half of the tail pointer, and we distinguish between two cases, as discussed in 2: either the enqueueing thread has yet to add the new node to the linked list and x_{tail} is still the last node, or the new node has been added, but the tail pointer hasn't been updated, meaning that x_{tail} is the second last node ($isSndLast$ is defined similarly to $isLast$).
In the **Dequeue** state, the dequeueing thread owns half of the head pointer, and the tail is as in the **Static** state.
Finally, the **Both** state is essentially a combination of the **Enqueue** and **Dequeue** states.

To track which state the queue is in, we use *tokens*. Tokens are defined using the exclusive resource algebra on the singleton set: $\text{Ex}()$. This resource algebra only has one valid element, and combining two elements will give the non-valid element $\bot$. Thus, if we own a particular token, then, upon opening the invariant, we can rule out certain states simply because they mention the token we own. We will use several tokens, each of which is the valid element of their own instance of $\text{Ex}()$. Different instances are distinguish between using ghost names. Hence, each token will be represented by a ghost name. As we did for the sequential specification, we group these ghost names into a tuple Q_γ, and write, for instance $\text{TokE } Q_\gamma$ to refer to the valid element of a particular instance. We proceed to explain the meaning of each of the tokens used in the invariant.

- $\text{ToknE } Q_\gamma$ represents that no threads are enqueueing.
- $\text{TokE } Q_\gamma$ represents that a thread is enqueueing.
- $\text{ToknD } Q_\gamma$ represents that no threads are dequeueing.
- $\text{TokD } Q_\gamma$ represents that a thread is dequeueing.
- $\text{TokBefore } Q_\gamma$ represents that an enqueueing thread has not yet added the new node to the linked list.
- $\text{TokBefore } Q_\gamma$ represents that an enqueueing thread has added the new node to the linked list, but not yet swung the tail.
- $\text{TokUpdated } Q_\gamma$ is defined as $\text{TokBefore } Q_\gamma * \text{TokBefore } Q_\gamma$, and represents that the queue is up to date.

Note: The concurrent specification for the two-lock Michael Scott Queue *can* be proven using the queue invariant 3.6.1, and the proof outline below will also be using this. However, a simpler (but arguably less intuitive) queue invariant was discovered. This simpler invariant is equivalent to 3.6.1 and has the benefit of being easier to work with in the mechanised proofs. Thus, in the mechanised proofs, the simpler variant is used. The simpler variant can be found in the appendix ▶**add appendix**◀.

With this, we can now give our definition of is_queue_conc. In the below, we let $\mathcal{N}$ be some namespace.

Definition 3.6.2 (Two-Lock M&S-Queue - is_queue_conc Predicate)**.**

$$\begin{aligned}
\text{is_queue_conc } \Psi\ v_q\ Q_\gamma \triangleq\, & \exists \ell_{queue}, \ell_{head}, \ell_{tail} \in Loc.\ \exists h_{lock}, t_{lock} \in Val. \\
& v_q = \ell_{queue} * \ell_{queue} \mapsto^{\square} ((\ell_{head}, \ell_{tail}), (h_{lock}, t_{lock})) * \\
& \boxed{queue_invariant \Phi \ell_{head} \ell_{tail} Q_\gamma}^{\mathcal{N}.queue} * \\
& isLock\ Q_\gamma.\gamma_{Hlock}\ h_{lock}\ (\text{TokD}\ \ Q_\gamma) * \\
& isLock\ Q_\gamma.\gamma_{Tlock}\ t_{lock}\ (\text{TokE}\ \ Q_\gamma).
\end{aligned}$$

In contrast to the sequential specification, the locks now protect TokE Q_γ and TokD Q_γ. The idea is that, when an enqueueing thread obtains t_{lock}, they will obtain the TokE Q_γ token, which allows them to conclude that the queue state is either **Static** or **Dequeue**. Similarly for a dequeueing thread. We now proceed to prove the specification using the above is_queue_conc predicate.

3.6.2 Proof outline

Firstly, we must show that is_queue_conc is persistent. This however follows from the fact that invariants are persistent, the *isLock* predicates are persistent, persistent points-to predicates are persistent, and persistency is preserved by $*$ and quantifications (rules: persistently-sep, persistently-$\wedge$, persistently-$\exists$).

The proofs structure for the specifications are largely similar to the sequential counterparts. The major difference is that we don't have access to the resources all the time; we must get them from the invariant. Further we also have to keep track of which state we are in. For the proof outlines below, these points will be the main focus.

Initialise

We first step through the first line which gives us the sentinel node of the linked list. Next, we must create the two locks. To create the two tokens that the locks must protect, we use the ghost-alloc rule twice, which gives us two ghost names, one for each of the tokens. We put the ghost names into a tuple Q_γ, and write TokE Q_γ and TokD Q_γ for the two ghost resources created by the ghost-alloc rule. We then create the locks, giving up the two tokens. Following this, we create the ℓ_{queue}, ℓ_{head}, and $\ell tail$ pointers. All that remains then is to prove the postcondition; the is_queue_conc predicate. The persistent points-to predicate we got when we stepped through the code, and the isLock predicates we got when we created the locks. So all that remains is the invariant. We create the *queue_invariant* in the **Static** state, most of which is analogous to the sequential specification. However, we will also need to supply the tokens required by the **Static** state. Thus, we allocate the four tokens ToknE Q_γ, ToknD Q_γ, TokBefore Q_γ, and TokBefore Q_γ in the same way we allocated TokE Q_γ and TokD Q_γ. We combine TokBefore Q_γ and TokBefore Q_γ to get TokUpdated Q_γ, and we now have all the tokens we need to create the *queue_invariant* in the **Static** state. To create the invariant from *queue_invariant*, we use the Inv-alloc rule (FUP).

Enqueue

We first step through the first line which gives us the new node x_{new}. We then acquire the tail lock t_{lock}, giving us TokE Q_γ. In the next line we must dereference the tail pointer, in order to get the tail node x_tail. This information, however, is inside the invariant. Invariant can only be opened if the expression being considered is atomic, but we can always make it atomic using the bind rule. Thus, we open the invariant, and since we have TokE Q_γ, we know that the queue is in state **Static** or **Dequeue**. In any case, we get that $\ell_{tail} \mapsto$ in x_{tail}, and that x_{tail} is the last node in the linked list. We can then dereference ℓ_{tail}, and must then close the invariant. We split up the points-to predicate $\ell_{tail} \mapsto$ in x_{tail} in two, which leaves us with two of $\ell_{tail} \mapsto^{\frac{1}{2}}$ in x_{tail}. We keep one of them, and use the other to close the invariant in the before case of state **Enqueue** or **Both**, depending on which state we opened the invariant into. By doing this, we give up TokE Q_γ, but we gain ToknE Q_γ and TokBefore Q_γ. We can now step to the point where x_{tail}'s out is updated to point to x_{new}. However, the points-to predicate concerning out x_{tail} isn't persistent, and is hence inside the invariant. We thus have to open the invariant again. Since we have ToknE Q_γ and TokBefore Q_γ, we know that we are in the before case of either state **Enqueue** or **Both**. We now get a different tail node, x'_{tail}, with $\ell_{tail} \mapsto^{\frac{1}{2}}$ in x'_{tail}. However, since we kept $\ell_{tail} \mapsto^{\frac{1}{2}}$ in x_{tail}, we can combine these, allowing us to conclude that in $x_{tail} =$ in x'_{tail}. Due to the structure of nodes (as described by isLL), we can further conclude that $x_{tail} = x'_{tail}$. This now gives us that out $x_{tail} \mapsto$ None, and we can perform the store, adding x_{new} to the linked list. We now wish to close the invariant in the after case of either state **Enqueue** or **Both**, giving up TokBefore Q_γ, and obtaining TokBefore Q_γ. When closing the invariant we shall pick as the abstract state $v :: xs_v$, where v is the enqueued value, and xs_v the abstract state we got when we opened the invariant. Note that in the pre-condition of the hoare-triple, we have Φv, hence we will be able to conclude $All(v :: xs_v)\Phi$. For the concrete state, we pick $x_{new} :: xs$, where xs is the concrete state we got when we opened the invariant. With these choices, we can close the invariant.

The next line swings the tail pointer to x_{new}. But to perform this store, we must first know that ℓ_{tail} points to something. This resource is inside the invariant, so we must open the invariant one last time. Due to our tokens, we know that we are in the after case of state **Enqueue** or **Both**. This time, we get some x''_{tail}, with $\ell_{tail} \mapsto^{\frac{1}{2}} x''_{tail}$, but we also get that x''_{tail} is only the second last node in the linked list. Hence there is some other node x'_{new}, which is the last node, with x''_{tail} pointing to it. As before, we use the points to predicate of ℓ_{tail} to get that $x''_{tail} = x_{tail}$. Since x_{tail} points to x_{new}, and x''_{tail} points to x'_{new}, we can further conclude that $x_{new} = x'_{new}$. Thus, we can perform the store, which now gives us that ℓ_{tail} points to x'_{new}; the last node in the linked list. With this, we can close the invariant in state **Static Dequeue**, giving up ToknE Q_γ and TokUpdated Q_γ, but getting TokE Q_γ. Finally, the code releases the lock, which we can do since we have TokE Q_γ. The postcondition only says True, so there is nothing left to prove.

Dequeue

We first acquire the lock, which gives us TokD Q_γ. Next, we must get the head node, by dereferencing ℓ_{head}. To do this, we must open the invariant. We open it in state **Static** or **Enqueue**, and conclude that there is some head node, x_{head}, with $\ell_{head} \mapsto x_{head}$. We perform the read, and take half of the points-to predicate. We then close the invariant in state **Dequeue** or **Both**, giving up TokD Q_γ, but gaining ToknD Q_γ. Next, we must find out what x_{head} points to by dereferencing out x_{head}. To perform this dereference, we must open the invariant. Using the token, we conclude that we open it in state **Dequeue** or **Both**. In any case, we get that there is some x'_{head} with $\ell_{head} \mapsto^{\frac{1}{2}} x'_{head}$. Using the fractional points-to predicate we kept from earlier, we can conclude that $x'_{head} = x_{head}$. We now perform a case analysis on the contents of the queue: xs_{queue}.

xs_{queue} **is empty**: In this case, we conclude that x_{head} points to None. We then perform the dereference of out x_{head}, giving us None. We close the invariant in state **Static** or **Enqueue**, giving up ToknD Q_γ and obtaining TokD Q_γ. We then step through the code, and since out x_{head} dereferenced to None, we take the if branch. We release the lock, giving up TokD Q_γ. The return value is None, so to finish the proof, we change the post-condition to prove the left disjunct.

xs_{queue} **is not empty**: We can now conclude that x_{head} points to some node x_{head_next}, whic is the first node in xs_{queue}. We perform the dereference, which gives us in x_{head_next}. We close the invariant in **Dequeue** or **Both**. We step through the code, taking the else branch. We extract the value from x_{head_next} (which we have access to since it is persistent). Next, we must swing ℓ_{head} to x_{head_next}, which requires that we know that ℓ_{head} points to something. Hence, we open the invariant in state **Dequeue** or **Both**, which gives us $\ell_{head} \mapsto^{\frac{1}{2}} x''_{head}$. We combine this with our half of the points-to predicate to conclude that $x''_{head} = x_{head}$. We then perform the store, giving us $\ell_{head} \mapsto$ in x_{head_next}. Closing the invariant now consists of removing the head element x_v from the abstract state xs_v, putting x_{head} into xs_{old}, removing x_{head_next} from xs_{queue} (which means that xs_{queue} is still reflected in xs_v) and letting x_{head_nxet} become the new x_{head}. In removing x_v from xs_v we may also extract Φx_v from $All xs_v \Phi$. With these changes, we can close the invariant in state **Static** or **Enqueue**, giving up ToknD Q_γ, and obtaining TokD Q_γ.
All that is left now is releasing the lock, which we do by giving up TokD Q_γ, and

we are left with the return value val x_{head_next}. We change the post-condition to prove the second disjunct. Since xs_{queue} was reflected in xs_v, and x_{head_next} was the head of xs_{queue}, and x_v the head of xs_v, then we can conclude that val $x_{head\_next} = \text{Some}\, x_v$. And since we had Φx_v, we can then finish the proof by choosing x_v as the witness in the post-condition and frame away Φx_v.

Discussing the need for xs_{old}

As mentioned in the observations, it is possible for the tail to lag one node behind the head. This insight lead to including the old nodes of the queue in the queue invariant. This addition manifests in the end of the proof of dequeue. When we open the invariant to swing ℓ_{head} to the x_{head_next}, we get that the entire linked list is xs. After performing the store, we can then close the invariant with the same xs that we opened the queue to, just written differently to signify that x_{head} is now "old", and x_{head_next} is the new head node. Because of this, we can supply the same predicate concerning the *tail* that we got when we opened the invariant, since this only mentions xs, which remains the same.

Had we not used an xs_{old} and essentially just "forgotten" old nodes, we couldn't have done this. Say that we defined xs as $xs = xs_{queue} +\!\!+ [x_{head}]$ instead. Then, once we have to close the invariant, we cannot supply xs, which we got when we opened the invariant. Our only choice (due to the fact that loc_{head} must point to x_{head_next}) is to close the invariant with $xs' = xs_{queue} = xs'_{queue} +\!\!+ [x_{head\_next}]$. However, clearly $xs' \neq xs$, so we cannot supply the same predicate concerning the *tail* that we got when opening the invariant, since this predicate talks about xs, not xs'. Now, if we opened the invariant in the state **Dequeue**, then we could conclude $isLast x_{tail} xs'$ from $isLast x_{tail} xs$, due to the relationship between xs and xs', and still be able close the invariant. However, if we opened the invariant in state **Both**, then we would need to assert $isSndLast x_{tail} xs'$ from $isSndLast x_{tail} xs$. This is however not provable, since $isSndLast x_{tail} xs$ allows for the case where xs'_{queue} is empty, which makes $xs' = [x_{head\_next}]$, disallowing us to prove $isSndLast x_{tail} xs'$.

3.7 Hocap-style Specification

When proving the concurrent specification, we were quite careful with tracking the state of the queue, and to some extent, even it's contents. The contents may have been existentially quantified, but through saving half a pointer, we could match up the contents of the queue between invariant openings. Given this precision in the proof, the reader may wonder if it is possible to give a more precise spec: one which is both concurrent and allows tracking of the contents of the queue. Indeed, this is possible, and we will explore such a spec in this section. We shall refer to this spec as a hocap-style spec (higher order concurrent abstract predicate) since the spec will be concurrent and parametrised by abstract predicates. This spec is more general than both the sequential and concurrent specs in the sense that those specs can be derived from the hocap-style spec. We prove this in section 3.8.

As before, we cannot simply parametrise the is_queue predicate with the abstract state of the queue, as we wish for it to be concurrent. So to allow clients to keep track of the contents of the queue, we will "split" the abstract state up in two parts, the authoritative view and the fragmental view. The client will then own the fragmental view, allowing them to keep track of the contents of the queue, whereas the is_queue predicate will own the authoritative view. We will in particular make sure that, if one has both the fragmental and authoritative views, then these agree on the abstract state of the queue. Further, it is only possible to update the abstract state of the queue (through the fup) if one possess both the authoritative and fragmental views.

We shall use the resource algebra $\textsc{Auth}((\textsc{Frac} \times \textsc{Ag}(\textit{List Val}))^?)$ to achieve the above. *List Val* is the abstract state. It is wrapped in the agreement RA, $\textsc{Ag}$, which ensures that if one owns two elements, then they agree on the abstract state. The fractional RA $\textsc{Frac}$, denotes how much of the fragmental view is owned; the fragmental view can be split up, which is handled by the clients. We collect $\textsc{Frac}$ and $\textsc{Ag}(\textit{List Val})$ in the product RA, whose elements are then pairs of fractions and abstract states. The option RA ?, makes the product RA unital which is required by the $\textsc{Auth}$ construction. $\textsc{Auth}$ is the authoritative resource algebra and gives us the authoritative and fragmental views, and governs that they can only be updated in unison.

As before, we collect the ghost names we will need in a tuple, this time of type *Qgnames*. It is similar to *ConcQgnames*, but with one additional ghost name: γ_{Abst} which is used for elements in the resource algebra we constructed above.

For an abstract state xs_v and a tuple Q_γ of type *Qgnames*, we shall use the notation $Q_\gamma \Mapsto_\bullet xs_v$ for the ownership assertion $\boxed{\bullet\,(1, \mathsf{ag}\ xs_v)}^{Q_\gamma.\gamma_{Abst}}$, meaning that the authoritative view of the abstract state associated with $Q_\gamma.\gamma_{Abst}$ is xs_v. Similarly we write $Q_\gamma \Mapsto_\circ xs_v$ for the assertion $\boxed{\circ\,(1, \mathsf{ag}\ xs_v)}^{Q_\gamma.\gamma_{Abst}}$.

With this, we now give the hocap-style specification, and explain it afterwards.

Lemma 3 (Two-Lock M&S-Queue Hocap Specification)**.**

$$
\begin{array}{ll}
\multicolumn{2}{l}{\exists\, \text{is\_queue} : \textit{Val} \to \textit{Qgnames} \to \textit{Prop}.} \\
 & \forall v_q, Q_\gamma.\ \text{is\_queue}\ \ v_q\ Q_\gamma \implies \Box\, \text{is\_queue}\ \ v_q\ Q_\gamma \\
\wedge & \{\textit{True}\}\ \text{initialize}()\ \{v_q.\exists Q_\gamma.\ \text{is\_queue}\ \ v_q\ Q_\gamma * Q_\gamma \Mapsto_\circ []\} \\
\wedge & \forall v_q, v, Q_\gamma, P, Q.\ \left(\forall xs_v.\ Q_\gamma \Mapsto_\bullet xs_v * P \Rrightarrow_{\mathcal{E}\setminus\mathcal{N}.i^{\uparrow}} \triangleright Q_\gamma \Mapsto_\bullet v :: xs_v * Q\right) \mathrel{-\!\!*} \\
 & \qquad\qquad \{\text{is\_queue}\ \ v_q\ Q_\gamma * P\}\ \text{enqueue}\ \ v_q\ v\, \{w.Q\} \\
\wedge & \forall v_q, Q_\gamma, P, Q.\ \left(\forall xs_v.\ Q_\gamma \Mapsto_\bullet xs_v * P \Rrightarrow_{\mathcal{E}\setminus\mathcal{N}.i^{\uparrow}} \triangleright \left(\begin{array}{l} (xs_v = [] * Q_\gamma \Mapsto_\bullet xs_v * Q\ \ \text{None}) \\ \vee \left(\begin{array}{l} \exists x_v, xs'_v.\ xs_v = xs'_v \mathbin{++} [x_v] * \\ Q_\gamma \Mapsto_\bullet xs'_v * Q\ (\text{Some}\, x_v) \end{array}\right) \end{array}\right)\right) \mathrel{-\!\!*} \\
 & \qquad\qquad \{\text{is\_queue}\ \ v_q\ Q_\gamma * P\}\ \text{dequeue}\ \ v_q\, \{v.Q\ v\}
\end{array}
$$

Firstly, we require is_queue to be persistent, giving us support for concurrent clients.

Next, the initialise spec gives clients an additional resource in the postcondition: the ownership of the fragmental view of the empty list, $Q_\gamma \Mapsto_\circ []$. As discussed above, this allows them to keep track of the contents of the queue.

Finally, the specs for enqueue and dequeue have been parametrised by two predicates P and Q. The clients get to pick P and Q, and the choice depends on what the client wishes to prove; P describes those resources that the client has before enqueue or dequeue, and Q the resources it will have after. Hence P is in the precondition and Q in the postcondition of the hoare triple. However, before the client gets access to the hoare triple for enqueue or dequeue they must prove a viewshift. This viewshift states how the abstract state of the queue will change as a result of running enqueue or dequeue, and further shows that P can be updated to Q. Note that the consequent**►or righthand-side?◄** of the viewshift contains a $\triangleright$. This signifies that the update in the abstract state is tied to a step in the code. The mask on the viewshift further disallows opening of invariants in the namespace $\mathcal{N}.i$. This is because, when proving the specs, we will use an invariant within this namespace. Thus, to be able to use the viewshift while our invariant is open, we must make sure the viewshift doesn't use our invariant (since invariants can only be opened once, before being closed).

It might seem a bit strange that the client has to prove that the abstract state can be updated, but remember that the client owns the fragmental view, and that both this and the authoritative view, which is owned by the queue, is needed to update the abstract state. When proving the viewshift, clients aren't updating the abstract state of the queue, they are merely showing that they can supply the fragmental view, allowing the abstract state to be updated. This then enables the queue to update the authoritative view of the abstract state (using the proved viewshift) in conjunction with updating the concrete view.

Exactly how the client supplies the fragmental view depends on what the client wants to achieve. We will see two options, when we derive the sequential and concurrent specs from this hocap spec.

3.7.1 The is_queue Predicate

Our definition of is_queue is almost the same as is_queue_conc, so we only mention the difference here. The full definition can be found in the appendix **►appendix◄**. The difference is that we no longer take the predicate Ψ, and the collection of ghost names is now of type $Qgnames$. Similarly, the queue invariant, *queue_invariant*, doesn't require the Ψ any more. Further, the assertion $All\ xs_v\ \Psi$ is changed to $Q_\gamma \Mapsto_\bullet xs_v$.

3.7.2 Proof outline

The proofs are largely similar to the concurrent spec, but now, instead of having to handle the Ψ predicate, we must work with the authoritative and fragmental views of the abstract state.
For initialise, we must additionally get ownership of the authoritative and fragmental view of the abstract state, both of which should state that it is empty. I.e. we must get $Q_\gamma \Mapsto_\bullet [] * Q_\gamma \Mapsto_\circ []$. We achieve this by own-op and and own-allocate, which requires us to show that $\bullet\,(1, \mathsf{ag}\ []) \cdot \circ\,(1, \mathsf{ag}\ []) \in \mathcal{V}$. This follows by the definitions of the resource algebras. We use $Q_\gamma \Mapsto_\bullet []$ to establish the queue invariant, and $Q_\gamma \Mapsto_\circ []$ to prove the post-condition.

For enqueue and dequeue, the only real changes are the points in the proof, where the concrete state of the queue is updated.

Enqueue We start by assuming the viewshift which allows us to update P to Q and $Q_\gamma \Rrightarrow_\bullet xs_v$ to $Q_\gamma \Rrightarrow_\bullet v :: xs_v$, for any xs_v. We must then prove the hoare triple for the expression enqueue v_q v. The only real change from the previous proof happens the second time we open the invariant; the first and third times, the abstract state doesn't change, hence we can simply frame away the newly added authoritative fragment concerning the abstract state, and continue as we did before. The second time we open the invariant, it is around the expression that adds the newly created node to the linked list (**►add line number◄**). When opening it, we get $Q_\gamma \Rrightarrow_\bullet xs_v$. As before, we also get all the resources to match up variables and step through the code, updating the concrete state. To close the invariant, we must make the same choice of abstract state as we did previously: $v :: xs_v$. This, however, requires us to obtain $Q_\gamma \Rrightarrow_\bullet v :: xs_v$. However, since we have $Q_\gamma \Rrightarrow_\bullet xs_v$ and P (from the precondition), we can apply the viewshift to obtain it, along with Q. This then allows us to close the invariant, and the proof proceeds as previously. At the end, we must also prove the postcondition Q, but this is no issue as we obtained that from the viewshift.

Dequeue We assume the viewshift and proceed as in the concurrent proof until we get to the second time we open the invariant, which is around the expression that reads head's next node. It is here that we figure out whether or not the queue is empty by doing case analysis on xs_{queue}.

In the case that the queue is empty, then the abstract state of the queue will not change. We thus apply the viewshift (we have P from the precondition and $Q_\gamma \Rrightarrow_\bullet xs_v$ from the invariant), which gives us the disjunct. The right disjunct states that the abstract state xs_v is non-empty, but since the abstract state is reflected in xs_{queue}, which *is* empty, then we know that the right disjunct is impossible. Hence we may assume the left disjunct. I.e. $xs_v = [] * Q_\gamma \Rrightarrow_\bullet xs_v * Q$ None. We now proceed as before, this time giving up $Q_\gamma \Rrightarrow_\bullet xs_v$ to close the invariant. After stepping through the code, we are left with proving the postcondition: Q None, which we got from the viewshift.

If the queue is not empty, then we do not apply the viewshift (as the abstract state doesn't change within this invariant opening), and simply continue as we did previously. The next time we open the invariant is around the expression that writes the new head to ℓ_{head}. It is this store that updates the abstract state of the queue, so it is within this invariant opening that we apply the viewshift (again, we have P from the precondition and $Q_\gamma \Rrightarrow_\bullet xs_v$ from the invariant). This time, we know that xs_{queue} is non-empty, and since xs_v is reflected in xs_{queue}, then we can conclude that the first disjunct is impossible, so the viewshift gives us $\exists x_v, xs'_v.\, xs_v = xs'_v ++ [x_v] * Q_\gamma \Rrightarrow_\bullet xs'_v * Q\ (\mathrm{Some}\, x_v)$. As before, we conclude that $\mathrm{Some}\, x_v$ is the return value (through the reflection between xs_{queue} and xs_v), and proceed to close the invariant, this time giving up $Q_\gamma \Rrightarrow_\bullet xs'_v$. Stepping through the code, we end up having to prove the post-condition $Q\ (\mathrm{Some}\, x_v)$, which we got from the viewshift.

3.8 Deriving Sequential and Concurrent specs from Hocap

In this section we show that we can derive the sequential and concurrent specifications from sections 3.3 and 3.5 from the Hocap-style specification. The derivations will need to show how to update the abstract state of the queue. To help with this, we use the following lemmas, both of which follow from the definitions of the involved resource algebras. The first shows the authoritative and fragmental views of the abstract state agree.

Lemma 4 (Abstract state agree)**.** $\forall xs_v, xs'_v.$
$Q_{\gamma H} \Mapsto_\bullet xs_v * Q_{\gamma H} \Mapsto_\circ xs'_v \vdash xs_v = xs'_v$

▶consider showing proof◀ The second shows that, if we own both the authoritative and fragmental views, we are allowed to update the abstract state to whatever we like.

Lemma 5 (Abstract state update)**.** $\forall xs_v, xs'_v, xs''_v.$
$Q_{\gamma H} \Mapsto_\bullet xs_v * Q_{\gamma H} \Mapsto_\circ xs'_v \vdash Q_{\gamma H} \Mapsto_\bullet xs''_v * Q_{\gamma H} \Mapsto_\circ xs''_v$

▶consider showing proof◀

3.8.1 Deriving Sequential spec

We define the is_queue_seq predicate as follows.

Definition 3.8.1 (Two-Lock M&S-Queue - is_queue_seq Predicate (Derive))**.**

$$\begin{aligned}\text{is\_queue\_seq } v_q\ xs_v\ Q_{\gamma S} \triangleq\ & \exists Q_{\gamma H} \in Qgnames. \\ & proj\_Qgnames\_seq\ Q_{\gamma H} = Q_{\gamma S} * \\ & \text{is\_queue } v_q\ Q_{\gamma H} * \\ & Q_{\gamma H} \Mapsto_\circ xs_v\end{aligned}$$

Here, $proj\_Qgnames\_seq\ Q_{\gamma H}$ simply creates an element of $SeqQgnames$, with ghost names matching those of $Q_{\gamma H}$. is_queue is the predicate from the hocap-style spec, hence we know that it is duplicable.

The **sequential initialise spec** follows almost directly from hocap-style initialise spec. They only differ in the post-condition. The post-condition in the hocap-style spec states $\exists Q_{\gamma H}.$ is_queue $v_q\ Q_{\gamma H} * Q_{\gamma H} \Mapsto_\circ []$, whereas we have to prove $\exists Q_{\gamma S}.$ is_queue_seq $v_q\ []\ Q_{\gamma S}$. If we choose $proj\_Qgnames\_seq\ Q_{\gamma H}$ for $Q_{\gamma S}$, then the equality in is_queue_seq becomes trivially true, and the postcondition we must prove follows from the hocap-style postcondition.

To prove the **sequential enqueue spec**, assume some v_q, v, xs_v, and $Q_{\gamma S}$. We must then show the hoare-triple concerning the expression: enqueue $v_q\ v$. To do this, we shall use the hocap-style spec for enqueue. This requires us to pick P and Q, and prove the resulting viewshift. We choose $P \triangleq Q_{\gamma H} \Mapsto_\circ xs_v$ and $Q \triangleq Q_{\gamma H} \Mapsto_\circ (v :: xs_v)$. Note that with this choice, the hoare triple we get after proving the viewshift almost matches the hoare triple we have to prove. The main thing we need is is_queue $v_q\ Q_{\gamma H}$ in the postcondition. However, since is_queue is persistent, and it is present in the precondition, we may assume it in the postcondition. Hence, all we have to prove is the viewshift:

$$\forall xs'_v.\, Q_\gamma \Mapsto_\bullet xs'_v * Q_{\gamma H} \Mapsto_\circ xs_v \Rrightarrow_{\mathcal{E}\setminus\mathcal{N}.i^\uparrow} \triangleright Q_\gamma \Mapsto_\bullet v :: xs'_v * Q_{\gamma H} \Mapsto_\circ (v :: xs_v)$$

So assume some xs'_v, $Q_\gamma \Mapsto_\bullet xs'_v$ and $Q_{\gamma H} \Mapsto_\circ xs_v$. We must then prove $\Rrightarrow_{\mathcal{E}\setminus\mathcal{N}.i^\uparrow} \triangleright Q_\gamma \Mapsto_\bullet (v :: xs'_v) * Q_{\gamma H} \Mapsto_\circ (v :: xs_v)$ By property 4, $xs_v = xs'_v$, hence, we can apply property 5 to update the authoritative and fragmental views to $(v :: xs_v)$, which is what we wanted.

We use a similar approach to above to prove the **sequential dequeue spec**. So we assume some v_q, xs_v, and $Q_{\gamma S}$, and must then prove the hoare-triple concerning the expression: dequeue v_q. This time, we use the hocap-style dequeue spec, with the following choices: $P \triangleq Q_{\gamma H} \Mapsto_\circ xs_v$, and $Q\ v \triangleq (xs_v = [] * v = \text{None} * Q_{\gamma H} \Mapsto_\circ xs_v) \vee (\exists x_v, xs'_v.\, xs_v = xs'_v \mathbin{++} [x_v] * v = \text{Some}\, x_v * Q_{\gamma H} \Mapsto_\circ xs'_v)$. In the same way as above, the hoare triple we get matches the one we have to prove (after a bit of manipulation). So we only have to prove the viewshift. First we conclude that the abstract states in the authoritative and fragmental views are equal. Then we do a case analysis on the abstract state, xs_v. If xs_v is empty, then we prove the left disjunct in the consequent of the viewshift, *without* updating the authoritative and fragmental views. If xs_v is non-empty, i.e. $xs_v = xs'_v \mathbin{++} [x_v]$ for some xs'_v and x_v, then we prove the right-side of the consequent in the viewshift, using property 5 to update the authoritative and fragmental views to the new abstract state (xs'_v).

3.8.2 Deriving Concurrent spec

Remember that we need the is_queue_conc predicate to be persistent, hence we cannot simply assert $Q_{\gamma H} \Mapsto_\circ xs_v$ as we did for is_queue_seq. Instead we will put it into an invariant. The predicate we will use looks as follows.

Definition 3.8.2 (Two-Lock M&S-Queue - is_queue_conc Predicate (Derive))**.**

$$\begin{aligned}\text{is\_queue\_conc}\ v_q\ xs_v\ Q_{\gamma C} \triangleq \exists Q_{\gamma H} \in Qgnames.\ & \\ & proj\_Qgnames\_conc\ Q_{\gamma H} = Q_{\gamma C} * \\ & \text{is\_queue}\ \ v_q\ Q_{\gamma H} * \\ & \boxed{\exists xs_v.\, Q_{\gamma H} \Mapsto_\circ xs_v * All\ xs_v\ \Psi}^{\mathcal{N}.c}\end{aligned}$$

Persistency of is_queue_conc follows by the persistency of is_queue and the fact that invariants and equalities are persistent.

The concurrent initialise spec follows from the hocap-style initialise spec, after allocating the invariant in the post-condition. We achieve this by applying the rules Ht-csq-vs and inv-alloc, to put the assertions $Q_{\gamma H} \Mapsto_\circ xs_v$ and $All\ []\Psi$ (which is trivially true) in the postcondition of the hocap style spec into an invariant.

Next, to prove the enqueue spec, we assume some v_q, v, and $Q_{\gamma C}$, and must then prove the hoare triple concerning the expression: enqueue $v_q\ v$. We specialise the hocap-style enqueue spec with $P \triangleq \Psi\ v$ and $Q \triangleq \mathsf{True}$. The hoare triple we get after proving the viewshift matches the hoare triple we must prove, except that its precondition is weaker: it doesn't mention the invariant or the equality. Hence, the hoare triple we have to prove simply follows by the rule of consequence. To prove the viewshift, we must supply the full fragmental view. When deriving the sequential spec, we had this available through P. But this time we shall get it by opening the invariant in is_queue_conc. Proving the viewshift is then similar to what we did for the sequential spec.

To derive the dequeue spec, we pick $P \triangleq \mathsf{True}$ and $Q\ v \triangleq v = \text{None} \vee (\exists x_v.\ v = \text{Some}\, x_v * \Psi\ x_v)$. Again, the hoare triple we get after proving the viewshift is exactly the hoare triple we must prove, except that its precondition is weaker. Hence, we only have to prove the viewshift. This is done analogously to the sequential case (i.e. case distinction on xs_v), except this time we get $Q_{\gamma H} \mapsto_\circ xs_v$ through the invariant, and in the case where xs_v is not empty, i.e. $xs_v = xs'_v +\!\!+ [x_v]$ for some xs'_v and x_v, we extract $\Psi\ x_v$ from $All\ xs_v\ \Psi$, and use this to prove the right disjunct in Q.

Chapter 4

The Lock-Free Michael Scott Queue

4.1 Introduction

In this chapter we will study the Lock-Free Michael Scott Queue. As with the two-lock version, the original implementation can be found in [1]. As the name "Lock-Free" suggests, the implementation doesn't rely on locks to achieve correct behaviour. Instead, it uses the atomic operation: CAS which we discussed in section 2.1 **▶Make sure you have discussed it◀**.

▶Introduce what we will go through in the chapter◀

4.2 Implementation

```
initialize ≜
  let node = ref (None, ref (None)) in
  ref (ref (node), ref (node))
```

```
enqueue Q value ≜
  let node = ref (Some value, ref (None)) in
  (rec loop_ =
    let tail = !(snd(!Q)) in
    let next = !(snd(!tail)) in
    if tail = !(snd(!Q)) then
      if next = None then
        if CAS (snd(!tail)) next node then
          CAS (snd(!Q)) tail node
        else loop ()
      else CAS (snd(!Q)) tail next; loop ()
    else loop ()
  ) ()
```

```
dequeue Q ≜
  (rec loop_ =
```

```
3        let head = !(fst(! Q)) in
4        let tail = !(snd(! Q)) in
5        let p = newproph in
6        let next = !(snd(! head)) in
7        if head = Resolve(!(fst(! Q)), p, ()) then
8          if head = tail then
9            if next = None then
10             None
11           else
12             CAS(snd(! Q)) tail next; loop ()
13         else
14           let value = fst(! next) in
15           if CAS (fst(! Q)) head next then
16             value
17           else loop ()
18       else loop ()
19       )()
```

▶mention the addition of the prophecy variable and the reason why (linearisation point)◀

4.3 Reachability

An important aspect in the correctness of the lock-free M&S-queue is which nodes a particular node is able to *reach* through the linked list (i.e. by following the chain of pointers), and how the head and tail pointers change during the lifetime of the queue.

Firstly, the underlying linked list still only ever grows, and it does so only at the end. Hence, the set of nodes that a given node can reach only ever grows. Further, all nodes can always reach the last node in the linked list.

Secondly, similarly to the two-lock variant, the correctness of the queue relies on the fact that the head and tail pointers are only ever swung towards the end of the linked list. That is, if a node can reach, say, the tail node at one point during the program, then it can reach any future tail nodes.

Thirdly, whereas it was possible for the tail node to lag behind the head node in the two-lock version, it is not possible in the lock-free version. Indeed, if such a scenario could happen, dequeue could crash! Consider the scenario where the head node is the last node in the linked list (hence the queue is empty), and the tail is lagging behind the head. If someone invokes dequeue, the check on line 8, which is supposed to detect an empty queue or a lagging tail will result to false, and hence, incorrectly, take the 'else' branch, which assumes that there is something to dequeue. But since the queue is empty, then *next* – the node after head – is None, and when we try to dereference *next* on line 14, we will crash. Therefore, our invariant must ensure that the tail never lags behind the head.

To capture these properties, we introduce two notions of reachability: concrete reachability and abstract reachability, which we introduce in the following sections. This way of modelling the queue was originally introduced in [2]. The presentation here borrows the same ideas, but the presentation differs in the sense that it is node-oriented instead of location-oriented. Moreover, we prove some additional properties of reachability which allows us to simplify the queue invariant slightly.

4.3.1 Concrete Reachability

We say that a node x_n in a linked list can concretely reach a node x_m when, if we start traversing succeeding nodes (by following the out and in pointers starting from x_n), we will eventually get to x_m. If this is the case, we write $x_n \rightsquigarrow x_m$. We allow for traversing zero nodes to reach x_m, which essentially means that all nodes can reach themselves. Formally, we define $x_n \rightsquigarrow x_m$ inductively as follows.

Definition 4.3.1 (Concrete Reachability)**.** $x_n \rightsquigarrow x_m \triangleq \text{in } x_n \mapsto^{\square} (\text{val } x_n, \text{out } x_n) *$ $(x_n = x_m \lor \exists x_p.\ \text{out } x_n \mapsto^{\square} \text{in } x_p * x_p \rightsquigarrow x_m)$

This definition firstly states that x_n is a node: $\text{in } x_n \mapsto^{\square} (\text{val } x_n, \text{out } x_n)$. Secondly, x_n is either the node to be reached x_m, or it has a succeeding node x_p, which can reach x_m. Note that the points-to propositions are all persistent, which mimics the fact that the linked list is only ever changed by appending new nodes to the end. **▶decide which of the following two sentences makes more sense◀** This in turn makes concrete reachability a persistent predicate. This in turn makes $x_n \rightsquigarrow x_m$ persistent for all x_n and x_m.

We proceed to prove some useful lemmas about concrete reachability.

Lemma 6 (reach-reflexive)**.** $x_n \rightsquigarrow x_n \ast\!\!\ast \text{ in } x_n \mapsto^{\square} (\text{val } x_n, \text{out } x_n)$

Proof. The $-\!\!*$ direction follows directly by the definition. To prove the $*\!\!-$ direction, it suffices to show $(x_n = x_n \lor \exists x_p.\ \text{out } x_n \mapsto^{\square} \text{in } x_p * x_p \rightsquigarrow x_n)$. Clearly, this follows as the left disjunction holds. □

Lemma 7 (reach-transitive)**.** $x_n \rightsquigarrow x_m -\!\!* x_m \rightsquigarrow x_o -\!\!* x_n \rightsquigarrow x_o$

Proof. We proceed by induction in $x_n \rightsquigarrow x_m$.

B.C. In the base case, $x_n = x_m$. We get to assume that $x_m \rightsquigarrow x_o$, and must prove $x_n \rightsquigarrow x_o$. Since $x_n = x_m$, we are done.

I.C. In the inductive case, we assume that x_n is a node that points to some x_p, which satisfies $x_m \rightsquigarrow x_o -\!\!* x_p \rightsquigarrow x_o$. Assuming $x_m \rightsquigarrow x_o$, we must prove $x_n \rightsquigarrow x_o$.
To prove $x_n \rightsquigarrow x_o$ we must first show that x_n is a node, which we have already established. Next, we must show that either $x_n = x_o$, or x_n steps to some x'_p which can reach x_o. We prove the second case by choosing our x_p for x'_p. Thus, we have to show $x_p \rightsquigarrow x_o$. This then follow by the induction hypothesis together with our assumption that $x_m \rightsquigarrow x_o$.

□

Lemma 8 (reach-from-is-node)**.** $x_n \rightsquigarrow x_m -\!\!* \text{ in } x_n \mapsto^{\square} (\text{val } x_n, \text{out } x_n)$

Proof. This follow immediately from the definition or concrete reachability. □

Lemma 9 (reach-to-is-node)**.** $x_n \rightsquigarrow x_m \mathrel{-\!\!*} \text{in } x_m \mapsto^{\square} (\text{val } x_m, \text{out } x_m)$

Proof. We proceed by induction in $x_n \rightsquigarrow x_m$. The base case follows by lemma 8 above. In the inductive case, we assume that x_n points to some x_p, which reaches x_m. Our induction hypothesis is in $x_m \mapsto^{\square} (\text{val } x_m, \text{out } x_m)$, which is also our proof obligation, so we are done. $\square$

Lemma 10 (reach-last)**.** $x_n \rightsquigarrow x_m \mathrel{-\!\!*} \text{out } x_n \mapsto \text{None} \mathrel{-\!\!*} x_n = x_m * \text{out } x_n \mapsto \text{None}$

Proof. Assuming $x_n \rightsquigarrow x_m$ and out $x_n \mapsto$ None, we must prove that $x_n = x_m$ and out $x_n \mapsto$ None. By $x_n \rightsquigarrow x_m$, we know that either $x_n = x_m$, or x_n points to some x_p and $x_p \rightsquigarrow x_m$. The first case immediately gives us everything we need to prove both goals. If we are in the second case, then we know that out $x_n \mapsto^{\square}$ in x_p. But by our initial assumption, out $x_n \mapsto$ None. Since in x_p is a location, then this is clearly a contradiction. $\square$

4.3.2 Abstract Reachability

As discussed, we wish to capture that if a node can reach either the head or tail node at one point during the program, then it can reach any future head or tail nodes. To do this we introduce the notion of abstract reachability. The idea is to introduce ghost variables that can "point" to nodes in the linked list, just as the head and tail pointers do. We shall write $\gamma \rightarrowtail x$ to mean that the ghost variable γ *abstractly points* to the node x. We shall construct the abstract points-to predicate so that we can update $\gamma \rightarrowtail x$ to $\gamma \rightarrowtail y$ only if x can concretely reach y, i.e. $x \rightsquigarrow y$. This additional restriction compared to the normal points-to predicate is what allows us to capture the property described above. We write $x \dashrightarrow \gamma$ to mean that the node x can *abstractly reach* the ghost variable γ. The idea is that, if we have established $x \dashrightarrow \gamma$, then no matter what node γ abstractly points to, for instance $\gamma \rightarrowtail y$, we can conclude $x \rightsquigarrow y$. This means that if we update $\gamma \rightarrowtail y$ in the future to, say $\gamma \rightarrowtail z$, then we can conclude that $x \rightsquigarrow z$.

To define the abstract points-to predicate and the abstract reach predicate, we create the following resource algebra: $\textsc{Auth}(\mathcal{P}(\mathit{Node}))$, where $\mathit{Node} = (\mathit{Loc} \times \mathit{Val}) \times \mathit{Loc}$. Here, the resource algebra $\mathcal{P}(\mathit{Node})$ denotes the set of subsets of *Node*, with union as the operation. The empty set is the unit element, meaning that $\mathcal{P}(\mathit{Node})$ is unital. We may now define abstract reach and abstract points-to as follows.

Definition 4.3.2 (Abstract Reach)**.** $x \dashrightarrow \gamma \triangleq \boxed{\circ\{x\}}^{\gamma}$

Definition 4.3.3 (Abstract Points to)**.** $\gamma \rightarrowtail x \triangleq \exists s.\, \boxed{\bullet\, s}^{\gamma} * \mathop{\Large\ast}_{x_m \in s} x_m \rightsquigarrow x$

One should think about sets $s \in \mathcal{P}(\mathit{Node})$ as specifying which nodes can abstractly reach a certain ghost variable. Due to how the authoritative resource algebra work, the assertion $\boxed{\circ\{x\}}^{\gamma}$ essentially states that x is *one* of the nodes that can reach the node that γ points to. This is because, when combining fragmental and authoritative elements, we get that the fragmental element is "smaller" than the authoritative. In this case, "smaller" amounts to "subset". Hence, $\boxed{\circ \{x\}}^{\gamma}$ means that, whatever the authoritative set is, it will contain the node x.

The authoritative set is existentially quantified as it can change over time, but whatever it is, we assert that all the nodes it contains *can* concretely reach the node that the ghost name is currently pointing to. This choice of definitions enables us to prove the properties we desired of abstract reachability above. The four lemmas below is all we need for abstract reachability when we prove the specification for the lock-free M&S-queue later.

Firstly, if we have a node, we may allocate some ghost variable γ which points to it and assert that the node can reach γ.

Lemma 11 (Abs-Reach-Alloc)**.** ***►is it a viewshift?◄*** $x \rightsquigarrow x \Rrightarrow (\exists \gamma.\, \gamma \rightarrowtail x * x \dashrightarrow \gamma)$

Proof. We assume $x \rightsquigarrow x$ and must show $\models\!\Rrightarrow \exists \gamma'.\, \gamma' \rightarrowtail x * x \dashrightarrow \gamma'$. By definition or the authoritative RA, the element $\bullet \{x\} \cdot \circ \{x\}$ is valid. Hence by the Ghost-alloc rule, we may get $\models\!\Rrightarrow \exists \gamma.\, \boxed{\bullet \{x\} \cdot \circ \{x\}}^{\gamma}$. By Upd-mono, we may strip away the update modality on the goal and the previous assertion. Thus, we must prove $\exists \gamma'.\, \gamma' \rightarrowtail x * x \dashrightarrow \gamma'$, and we have that $\exists \gamma.\, \boxed{\bullet \{x\} \cdot \circ \{x\}}^{\gamma}$. We use γ as the witness in the goal meaning we must prove $\gamma \rightarrowtail x * x \dashrightarrow \gamma$. We can split the ownership of the authoritative and fragmental parts up using Own-op, giving us $\boxed{\bullet \{x\}}^{\gamma}$ and $\boxed{\circ \{x\}}^{\gamma}$. The latter assertion is equivalent to $x \dashrightarrow \gamma$, which matches the second obligation in the goal. To prove the first obligation, we must give some set as witness, and show that all nodes in the set can reach x. We of course choose $\{x\}$ as the witness, and must then prove that $x \rightsquigarrow x$, which we assumed in the beginning. □

The second lemma allows us to get a *concrete* reachability predicate out of an abstract one. If a ghost name γ_m currently points abstractly to some node x_m, then any node that can abstractly reach γ, can also *concretely* reach x_m.

Lemma 12 (Abs-Reach-Concr)**.** $x_n \dashrightarrow \gamma_m \mathrel{-\!\!*} \gamma_m \rightarrowtail x_m \mathrel{-\!\!*} x_n \rightsquigarrow x_m * \gamma_m \rightarrowtail x_m$

Proof. Assuming $x_n \dashrightarrow \gamma_m$ and $\gamma_m \rightarrowtail x_m$, we must show $x_n \rightsquigarrow x_m$ without "consuming" $\gamma_m \rightarrowtail x_m$. From $\gamma_m \rightarrowtail x_m$ we can deduce that there is some set s so that $\boxed{\bullet\, s}^{\gamma_m}$ and $\mathop{\ast}_{x' \in s} x' \rightsquigarrow x_m$. Since we own both $\boxed{\bullet\, s}^{\gamma_m}$ and $\boxed{\circ \{x_n\}}^{\gamma_m}$ (from $x_n \dashrightarrow \gamma_m$), we may conclude that their product is valid, which in our instantiation of the authoritative RA equates to $x_n \in s$. We may now frame away the second part of the goal, $\gamma_m \rightarrowtail x_m$, using $\boxed{\bullet\, s}^{\gamma_m}$ and $\mathop{\ast}_{x' \in s} x' \rightsquigarrow x_m$. Note that we get to keep the latter assertion as reach is persistent. Thus, from that assertion and by $x_n \in s$, we can deduce that $x_n \rightsquigarrow x_m$, which is what we had to prove. □

We can also go the other way, and get an abstract reachability predicate out of a concrete one. If a ghost variable γ_m points abstractly to some node x_m, and a node x_n can *concretely* reach x_m, then we may deduce that x_n can *abstractly* reach γ_m, meaning that x_n can reach any node that γ_m will ever point to in the future.

Lemma 13 (Abs-Reach-Abs)**.** $x_n \rightsquigarrow x_m \mathrel{-\!\!*} \gamma_m \rightarrowtail x_m \Rrightarrow x_n \dashrightarrow \gamma_m * \gamma_m \rightarrowtail x_m$

Proof. Assuming $x_n \rightsquigarrow x_m$ and $\gamma_m \rightarrowtail x_m$ we must conclude $\models\!\!\Rrightarrow x_n \dashrightarrow \gamma_m$. From $\gamma_m \rightarrowtail x_m$ we know that there is some set s so that $\boxed{\bullet\, s}^{\gamma_m}$ and $\mathop{\Large\ast}_{x' \in s} x' \rightsquigarrow x_m$. There are now two cases to consider: either $x_n \in s$ or $x_n \notin s$.

$x_n \in s$ By the definition of our authoritative RA, if a set y is a subset of s, then we may update our ghost resources to obtain ownership of the fragment y. In our case, since $x_n \in s$, we may update our resources to additionally get $\boxed{\circ\, \{x_n\}}^{\gamma_m}$, which is exactly what we wanted. Since we still have $\boxed{\bullet\, s}^{\gamma_m}$, we can also prove $\gamma_m \rightarrowtail x_m$.

$x_n \notin s$ In this case we may update $\boxed{\bullet\, s}^{\gamma_m}$ so that the set also includes x_n. The reason we may do this, is because, according to the $\mathcal{P}(\mathit{Node})$ RA, we may update a set X to Y, as long as $X \subseteq Y$. Thus, we can update our resource to get $\boxed{\bullet\, \{x_n\} \cup s}^{\gamma_m}$. As in the previous case, we can further get $\boxed{\circ\, \{x_n\}}^{\gamma_m}$ out of this, which we use to frame away the goal $x_n \dashrightarrow \gamma_m$.
To prove $\gamma_m \rightarrowtail x_m$, we use the set $\{x_n\} \cup s$, and immediately frame away the authoritative part, which we owned. We are left with having to prove $\mathop{\Large\ast}_{x' \in \{x_n\} \cup s} x' \rightsquigarrow x_m$. However, by $\mathop{\Large\ast}_{x' \in s} x' \rightsquigarrow x_m$ and our assumption that $x_n \rightsquigarrow x_m$, we can easily conclude this.

□

The final lemma allows us update abstract pointers. As discussed above, we will require that whatever node we update the pointer to is a successor of the current node. That is, if a ghost variable γ_m currently points to x_m, then we must show that x_m can reach x_o, before we can update γ to point abstractly to x_o. After the update we additionally get that x_o can abstractly reach γ.

Lemma 14 (Abs-Reach-Advance)**.** $\gamma_m \rightarrowtail x_m \mathrel{-\!\!*} x_m \rightsquigarrow x_o \Rrightarrow \gamma_m \rightarrowtail x_o * x_o \dashrightarrow \gamma_m$

Proof. Assuming $\gamma_m \rightarrowtail x_m$ and $x_m \rightsquigarrow x_o$ we must prove $\models\!\!\Rrightarrow \gamma_m \rightarrowtail x_o * x_o \dashrightarrow \gamma_m$. From $\gamma_m \rightarrowtail x_m$, we get some set s so that $\boxed{\bullet\, s}^{\gamma_m}$ and $\mathop{\Large\ast}_{x' \in s} x' \rightsquigarrow x_m$. As we did in previous proof (lemma 13), we update $\boxed{\bullet\, s}^{\gamma_m}$ so that the set additionally contains x_o. Thus, we get $\boxed{\bullet\, \{x_o\} \cup s}^{\gamma_m}$. From this, we may extract ownership of the fragmental part: $\boxed{\bullet\, \{x_o\}}^{\gamma_m}$, which we use to prove the second part of the goal.
We are thus left with proving $\gamma_m \rightarrowtail x_o$. We use $\{x_o\} \cup s$ as witness for the authoritative set, and immediately frame away the ownership assertion of the authoritative part. We are left with proving $\mathop{\Large\ast}_{x' \in \{x_o\} \cup s} x' \rightsquigarrow x_o$. We already know that $\mathop{\Large\ast}_{x' \in s} x' \rightsquigarrow x_m$ and $x_m \rightsquigarrow x_o$. Thus, by transitivity of reach (lemma 7), we may conclude $\mathop{\Large\ast}_{x' \in \cup s} x' \rightsquigarrow x_o$. Thus, we are done if we can prove that $x_o \rightsquigarrow x_o$, which by 6 amount to showing that x_o is a node. However, since $x_m \rightsquigarrow x_o$, then, by 9, we know that this is the case. □

4.4 Specification for Lock-Free M&S-Queue

From the perspective of a client, the two-lock M&S-Queue and the lock-free M&S-Queue should behave similarly – they should both behave as a concurrent queue. Hence, in this section, we will prove specifications that are almost identical to those we proved for the two-lock M&S-queue; a sequential, a concurrent, and a hocap-style spec.

As we showed in the previous chapter, the sequential and concurrent specifications can be derive from the hocap-style spec *without* referring to the actual implementation. Thus, in this chapter we will only focus on proving the hocap-style spec – the derivations of the sequential and concurrent specs will be practically identical to that of the last chapter.

The only two differences between the hocap spec we prove for the lock-free version compared to the two-lock version (lemma 3) is the collection of ghost names, and the fact that the expressions in our hoare-triples – initialize, enqueue, and dequeue – refer to the lock-free versions from section 4.2.
The collection $Qgnames$ will contain γ_{Abst} whose purpose is the same as before; to keep track of the abstract state of the queue. Additionally, we will have γ_{Head}, γ_{Tail}, and γ_{Last}, which will abstractly point to the head, tail, and last node, respectively.

4.4.1 The is_queue predicate

We will again be needing an invariant to make the predicate persistent. The invariant we define has some commonalities with the invariant we used for the two-lock variant, but it incorporates the differences we discussed earlier in the chapter. In particular, it is important for the correctness of the queue that the tail doesn't lag behind the head. As such, our invariant will not allow for this behaviour. This has the extra implication that the head node is always the oldest node, meaning that we do not need to keep track of older nodes, xs_{old}.

Unlike the two-lock variant, we assert the existence of an additional node x_{last}, which invariantly is the last (newest added) node in the linked list. This helps us reason about where the head and tail nodes are located; enqueue distinguishes between the cases where the tail is last and not last, and similarly for dequeue and head.
In this way, x_{head} is the first node, x_{last} is the last node, and x_{tail} either lies somewhere in between, is one of them, or, in the case where the queue is empty, is both of them. To force this structure, we use our abstract reachability predicate from the previous section.
We proceed to define the invariant.

Definition 4.4.1 (Lock-Free M&S-Queue Invariant).

$$
\begin{aligned}
& queue\_invariant\ \ell_{head}\ \ell_{tail}\ Q_\gamma \triangleq \\
& \exists xs_v.\ Q_\gamma \Rrightarrow_\bullet xs_v * && \textit{(abstract state)} \\
& \exists xs, xs_{queue}, x_{head}, x_{tail} x_{last}. && \textit{(concrete state)} \\
& xs = xs_{queue} +\!\!+ [x_{head}] * \\
& \text{isLL}\ \ xs * \\
& isLast\ x_{last}\ xs \\
& proj\_val\ xs_{queue} = wrap\_some\ xs_v * \\
& \ell_{head} \mapsto \text{in}\ x_{head} * \\
& \ell_{tail} \mapsto \text{in}\ x_{tail} * \\
& Q_\gamma.\gamma_{Head} \rightarrowtail x_{head} * x_{head} \dashrightarrow Q_\gamma.\gamma_{Tail} * \\
& Q_\gamma.\gamma_{Tail} \rightarrowtail x_{tail} * x_{tail} \dashrightarrow Q_\gamma.\gamma_{Last} * \\
& Q_\gamma.\gamma_{Last} \rightarrowtail x_{last}
\end{aligned}
$$

The is_queue predicate is now quite simple: it states that the value representing the queue is a location which points persistently to a pair of locations, the head and tail pointers, which satisfy the invariant we defined above.

Definition 4.4.2 (Lock-Free M&S-Queue - is_queue Predicate).

$$
\begin{aligned}
\text{is\_queue}\ v_q\ Q_\gamma \triangleq \exists \ell_{queue}, \ell_{head}, \ell_{tail} \in \mathit{Loc}.& \\
v_q = \ell_{queue} * \ell_{queue} \mapsto^{\square} (\ell_{head}, \ell_{tail}) *& \\
\boxed{queue\_invariant \ell_{head} \ell_{tail} Q_\gamma}^{\mathcal{N}.queue}&
\end{aligned}
$$

4.4.2 Proof outline

We instantiate the specification with our definition of is_queue (4.4.2). By the definition of is_queue we easily show that is_queue is persistent. What remains to be shown is the specifications for initialize, enqueue, and dequeue. Both enqueue and dequeue has code that attempt to swing the tail pointer forward (for enqueue, lines 9 and 11, and for dequeue, line 12). These all behave similarly, so we additionally prove a specification for swinging the tail.

Initialise

Lemma 15 (Lock-Free M&S-Queue Specification - Initialise).

$$\{\mathsf{True}\}\ \text{initialize()}\ \{v_q.\exists Q_\gamma.\ \text{is\_queue}\ \ v_q\ Q_\gamma * Q_\gamma \Rrightarrow_\circ []\}$$

Proof. We first step through line 2 which creates a new node: $x_1 = (\ell_{1\_\text{in}}, \text{None}, \ell_{1\_\text{out}})$, with $\ell_{1\_\text{out}} \mapsto \text{None}$ and $\ell_{1\_\text{in}} \mapsto (\text{None}, \ell_{1\_\text{out}})$, the latter of which we make persistent. Next, we step through line 3 which gives us some locations ℓ_{head}, ℓ_{tail}, and ℓ_{queue}, with $\ell_{head} \mapsto x_1$ and $\ell_{tail} \mapsto x_1$, and finally $\ell_{queue} \mapsto (\ell_{head}, \ell_{tail})$. As we did for the two-lock version, we allocate an empty abstract queue, giving us some ghost name γ_{Abst} that we put into Q_γ, and the resources $Q_\gamma \Rrightarrow_\bullet [] * Q_\gamma \Rrightarrow_\circ []$.

To allocate the invariant, we must additionally obtain abstract reach propositions. Since x_1 is a node, we may use lemma 6 to conclude $x_1 \rightsquigarrow x_1$. We can now use lemma 11 three times, giving us ghost names γ_{Head}, γ_{Tail}, γ_{Last} which we again put into Q_γ, and the resources

$$Q_\gamma.\gamma_{Head} \rightarrowtail x_1 * Q_\gamma.\gamma_{Tail} \rightarrowtail x_1 * x_1 \dashrightarrow Q_\gamma.\gamma_{Tail} * Q_\gamma.\gamma_{Last} \rightarrowtail x_1 * x_1 \dashrightarrow Q_\gamma.\gamma_{Last}$$

We now have all the resources we need to allocate the invariant with the head, tail, and last node being x_1.
With the invariant allocated, proving the post-condition becomes straightforward.

□

Swing Tail

The specification we wish to prove is the following.

Lemma 16 (Swing Tail)**.** $\forall \ell_{head}, \ell_{tail}, x_{tail}, x_{newtail}, Q_\gamma.$

$$\left\{\boxed{queue\_invariant\ \ell_{head}\ \ell_{tail}\ Q_\gamma}^{\mathcal{N}.queue} * x_{tail} \rightsquigarrow x_{newtail} * x_{newtail} \dashrightarrow Q_\gamma.\gamma_{Last}\right\}$$

$$\mathsf{CAS}\ \ell_{tail}\ \text{ in }\ x_{tail}\ \text{ in }\ x_{newtail}$$

$$\{v.v = \mathsf{true} \lor v = \mathsf{false}\}$$

Proof. The rule for CAS demands that we have a points-to predicate for ℓ_{tail}. This is available inside the invariant, so we proceed to open it. This tells us that there is some x'_{tail} so that $\ell_{tail} \mapsto x'_{tail}$. We consider both cases of the CAS:

Case CAS succeeds. It must then have been the case that in $x'_{tail} =$ in x_{tail}. Since we have $x_{tail} \rightsquigarrow x_{newtail}$, we know that x_{tail} is a node (lemma 8). From the invariant, we additionally got that $Q_\gamma.\gamma_{Tail} \rightarrowtail x'_{tail}$ and $x_{head} \dashrightarrow Q_\gamma.\gamma_{Tail}$, which by lemma 12 means that $x_{head} \rightsquigarrow x'_{tail}$. We can thus also conclude that x'_{tail} is a node (lemma 9). So since both x_{tail} and x'_{tail} are nodes, and in $x'_{tail} =$ in x_{tail}, then it must be that $x_{tail} = x'_{tail}$. In other words, we have now know that $Q_\gamma.\gamma_{Tail} \rightarrowtail x_{tail}$.
Since the CAS succeeded, we now have that $\ell_{tail} \mapsto$ in $x_{newtail}$. Since the invariant demands that $Q_\gamma.\gamma_{Tail}$ and ℓ_{tail} agree on the node they point to, we must update $Q_\gamma.\gamma_{Tail} \rightarrowtail x_{tail}$ to $Q_\gamma.\gamma_{Tail} \rightarrowtail x_{newtail}$. We can do this using lemma 14 as we assumed $x_{tail} \rightsquigarrow x_{newtail}$. With this, we can close the invariant again, using $x_{newtail}$ as the tail node.
The CAS evaluates to true which we use to prove the first disjunct of the post-condition.

Case CAS Fails. Since the CAS failed, nothing was updated, and we can close the invariant again with the same resources we got out of it. The CAS evaluates to false, hence we can prove the second disjunct in the post-condition.

□

Enqueue

Lemma 17 (Lock-Free M&S-Queue Specification - Enqueue)**.**

$$\forall v_q, v, Q_\gamma, P, Q.\ \left(\forall xs_v.\ Q_\gamma \Mapsto_\bullet xs_v * P \Rrightarrow_{\mathcal{E}\setminus\mathcal{N}.i\uparrow} \triangleright Q_\gamma \Mapsto_\bullet v :: xs_v * Q\right) \twoheadrightarrow$$
$$\{\text{is\_queue}\ \ v_q\ Q_\gamma * P\}\ \text{enqueue}\ \ v_q\ v\ \{w.Q\}$$

Proof. We assume the viewshift and proceed to prove the hoare triple. By definition of is_queue, we know that the queue v_q is a location ℓ_{queue} and there are locations ℓ_{head} and ℓ_{tail} so that

$$\ell_{queue} \mapsto^{\square} (\ell_{head}, \ell_{tail}) * \boxed{queue\_invariant \ell_{head} \ell_{tail} Q_{\gamma}}^{\mathcal{N}.queue} \tag{4.1}$$

We first step through line 2 which creates a new node x_{new}, so that

$$\text{in } x_{new} \mapsto^{\square} (\text{Some}\, v, \text{out } x_{new}) \tag{4.2}$$

$$\text{out } x_{new} \mapsto \text{None} \tag{4.3}$$

The next line is the beginning of the looping function. We proceed by löb induction, which allows us to assume the hoare triple we wish to prove *later*. This means that, if we reach a recursive call, we will have the hoare triple that we must prove – the later will be immediately stripped when we apply () and "step into" the recursive function.
Line 4 first dereferences to the tail pointer ℓ_{tail} using the resources in 4.1. We open the invariant to obtain the points-to predicate concerning ℓ_{tail}. We get that ℓ_{tail} points to some x_{tail}. Using the resources from the invariant, we may conclude the following persistent information:

$$x_{tail} \dashrightarrow Q_{\gamma}.\gamma_{Last} \tag{4.4}$$

$$\text{in } x_{tail} \mapsto^{\square} (\text{val } x_{tail}, \text{out } x_{tail}) \tag{4.5}$$

The first part is directly from the invariant, and the second we may derive using lemmas 12 and 9. We perform the load, and close the invariant.
The next line (line 5) finds out what x_{tail} points to. Using 4.5 we step to $!(\text{out } x_{tail})$. The points-to predicate required to perform this dereference is owned by the invariant (as it might be non-persistent), so we open the invariant again. We get that there is some x_{last}, with $Q_{\gamma}.\gamma_{Last} \rightarrowtail x_{last}$. From this, 4.4, and lemma 12 we conclude $x_{tail} \rightsquigarrow x_{last}$. This gives us two cases to consider: either x_{tail} *is* x_{last} (meaning that x_{tail} is not lagging behind), or it points to some node x_{tail_next} which can reach x_{last} (meaning that x_{tail} is lagging behind).

Case $x_{tail} = x_{last}$. Since we had $isLast x_{last} xs$, we know that x_{tail} is the last node in the linked list, hence it points to None. We perform the load which sets *next* to None, and close the invariant.
We proceed to the consistency check on line 6. As before, the points-to predicate for ℓ_{tail} is in the invariant, so we open it. We get $\ell_{tail} \mapsto x'_{tail}$, for some x'_{tail}. Using this, we perform the deference and close the invariant. The branch taken now depends on whether or not x'_{tail} is consistent with x_{tail}. In case they aren't, we take the 'else branch on line 12, which simply consists of a recursive call to the looping function. We are done by the induction hypothesis (from the löb induction).
If they are consistent, we take the 'then' branch and step to line 7. Here we check whether or not *next* is None. We already know this is the case, so we proceed to line 8. This consists of a CAS instruction which attempts to add x_{new} to the linked list. The CAS will succeed if and only if ℓ_{tail}

still points to None. We open the invariant to gain access to the relevant points-to predicate. Similarly to what we did earlier, we apply lemma 12 to conclude $x_{tail} \rightsquigarrow x'_{last}$, where x'_{last} is the current last node of the linked list (according to the invariant). As before, we perform case analysis on $x_{tail} \rightsquigarrow x'_{last}$.

Case $x_{tail} = x'_{last}$. We now know that x_{tail} is still the last node in the linked list, hence out $x_{tail} \mapsto$ None, and the CAS will succeed. This instruction makes x_{tail} point to x_{new}, which essentially adds it to the linked list. Thus, the value in x_{new} becomes enqueued. In other words, this is a linearisation point, so we must apply the viewshift. We instantiate the viewshift with the abstract state of the queue xs_v from the invariant opening, and supply $Q_\gamma \Rrightarrow_\bullet xs_v$ from the invariant and the P from the pre-condition. We hence get $Q_\gamma \Rrightarrow_\bullet v :: xs_v$ and Q.
When closing the invariant, we use $(v :: xs_v)$ for the abstract state, $(x_{new} :: xs)$ for the concrete state, $x_{new} :: xs_{queue}$ for the queue, and we take x_{new} to be the last node. The head and tail nodes remain the same. This means we give up $Q_\gamma \Rrightarrow_\bullet v :: xs_v$ that we got from the viewshift, and the points-to predicate 4.3 (used to assert isLL$(x_{new} :: xs)$). The only thing left to prove is $Q_\gamma.\gamma_{Last} \rightarrowtail x_{new}$. From the invariant opening, we have $Q_\gamma.\gamma_{Last} \rightarrowtail x_{tail}$. Since $x_{tail} \rightsquigarrow x_{new}$, we may apply lemma 14 to update the abstract points-to resource to $Q_\gamma.\gamma_{Last} \rightarrowtail x_{new}$ and additionally obtain $x_{new} \dashrightarrow Q_\gamma.\gamma_{Last}$. With this, we can close the invariant, and step to line 9. This line attempts to swing the tail, so we apply our swing-tail lemma (lemma 16) by supplying our invariant, $x_{tail} \rightsquigarrow x_{new}$, and $x_{new} \dashrightarrow Q_\gamma.\gamma_{Last}$. This tells us that the CAS is safe, and it either succeeds or fails. The resulting value is the returned value of the enqueue function, but since the post-condition is simply Q, which we own, we are done.

Case out $x_{tail} \mapsto^\square$ in $x_{tail\_next} * x_{tail\_next} \rightsquigarrow x_{last}$. Since x_{tail} doesn't point to None, the CAS will fail. We close the invariant and step to line 10. We finish by applying the induction hypothesis.

Case out $x_{tail} \mapsto^\square$ in $x_{tail\_next} * x_{tail\_next} \rightsquigarrow x_{last}$. Using this we perform the load, which sets *next* to in x_{tail_next}. Before closing the invariant, we apply lemma 13 with $x_{tail\_next} \rightsquigarrow x_{last}$ and $Q_\gamma.\gamma_{Last} \rightarrowtail x_{last}$ to obtain $x_{tail\_next} \dashrightarrow Q_\gamma.\gamma_{Last}$. We now proceed to close the invariant. Next, we reach the consistency check. We handle it similarly to the previous case: open the invariant, get some x'_{tail}, close the invariant, and in case of inconsistency, apply induction hypothesis. If the nodes are consistent, we step to line 7. This time, the check will fail as *next* is in x_{tail_next} which is not None. Hence we step to line 11 which attempts to swing the tail pointer. We here apply lemma 16 which we can do as we own the invariant, $x_{tail} \rightsquigarrow x_{tail\_next}$, and $x_{tail\_next} \dashrightarrow Q_\gamma.\gamma_{Last}$. We step through to the recursive call and finish by applying the induction hypothesis.

$\square$

Dequeue

Lemma 18 (Lock-Free M&S-Queue Specification - Enqueue).

$$\forall v_q, Q_\gamma, P, Q. \quad \left(\forall xs_v.\, Q_\gamma \Rrightarrow_\bullet xs_v * P \Rrightarrow_{\mathcal{E}\setminus\mathcal{N}.i\uparrow} \triangleright \left(\begin{array}{l} (xs_v = [] * Q_\gamma \Rrightarrow_\bullet xs_v * Q \;\; \text{None}) \\ \vee \left(\begin{array}{l} \exists x_v, xs'_v.\, xs_v = xs'_v +\!\!+ [x_v] * \\ Q_\gamma \Rrightarrow_\bullet xs'_v * Q \;(\text{Some}\, x_v) \end{array} \right) \end{array} \right) \right) \twoheadrightarrow$$
$$\{\text{is\_queue}\;\; v_q\;\; Q_\gamma * P\}\;\text{dequeue}\;\; v_q\;\{v.Q\;\; v\}$$

Proof. We assume the viewshift and must prove the hoare triple. As before, we know from is_queue that the queue v_q is a location ℓ_{queue} and there are locations ℓ_{head} and ℓ_{tail} so that

$$\ell_{queue} \mapsto^{\square} (\ell_{head}, \ell_{tail}) * \boxed{queue\_invariant \ell_{head} \ell_{tail} Q_\gamma}^{\mathcal{N}.queue} \tag{4.6}$$

The body of dequeue is the loop, so we immediately apply löb induction. We step through the function application, and into the looping function to line 3. This line dereferences ℓ_{head}, so we open the invariant to access the associated points-to predicate. We obtain that ℓ_{head} points to some x_{head}, meaning the load results to in x_{head}. We also derive the following information

$$\text{in}\;\; x_{head} \mapsto^{\square} (\text{val}\;\; x_{head}, \text{out}\;\; x_{head}) \tag{4.7}$$
$$x_{head} \dashrightarrow Q_\gamma.\gamma_{Head} \tag{4.8}$$
$$x_{head} \dashrightarrow Q_\gamma.\gamma_{Tail} \tag{4.9}$$
$$x_{head} \dashrightarrow Q_\gamma.\gamma_{Last} \tag{4.10}$$

From the abstract points-to predicates from the invariant and lemma 12 we get that $x_{head} \rightsquigarrow x_{tail}$, so by lemma 8, we know that x_{head} is a node. This shows 4.7. By reflexivity of reach (lemma 6) we additionally know that $x_{head} \rightsquigarrow x_{head}$. Lemma 13 then gives us 4.8 and 4.9.
Lastly, we use lemma 12 to deduce that $x_{tail} \rightsquigarrow x_{last}$. By transitivity of reach (lemma 7) we have that $x_{head} \rightsquigarrow x_{last}$, which we use with lemma 13 to get 4.10. We now close the invariant and step to line 4 which attempts to read ℓ_{tail}. We open the invariant, which tells us that ℓ_{tail} points to some x_{tail} (not necessarily the same as the previous invariant opening), and we perform the load. From the abstract points-to and reach predicates from the invariant together with 4.9 and lemmas 12, 9, and 13 we get the following

$$\text{in}\;\; x_{tail} \mapsto^{\square} (\text{val}\;\; x_{tail}, \text{out}\;\; x_{tail}) \tag{4.11}$$
$$x_{head} \rightsquigarrow x_{tail} \tag{4.12}$$
$$x_{tail} \dashrightarrow Q_\gamma.\gamma_{Tail} \tag{4.13}$$
$$\tag{4.14}$$

We close the invariant and step to line 5. This line creates our *prophecy variable* p, which will be resolved on line 7. This allows us to reason about the result of the consistency check: we will later show that the expression associated with p (i.e. $!(\text{fst}(!\,Q))$) evaluates to some value v_p, but we can already now case on whether v_p will be equal to in x_{head} – the left hand side of the equality check on line 7.

Case in $x_{head} = v_p$. We continue to line 6, which finds out what x_{head} points to. As x_{head} could be the last node in the linked list, we don't have the relevant points-to predicate. We therefore open the invariant. We get the three nodes x'_{head}, x'_{tail}, and x_{last}. Specifically, x_{last} is the last node in the linked list, and

$$Q_\gamma.\gamma_{Last} \rightarrowtail x_{last} \tag{4.15}$$

Combining this with 4.10 and lemma 12 we conclude $x_{head} \rightsquigarrow x_{last}$. This gives us two cases to consider: either $x_{head} = x_{last}$ or x_{head} points to some x_{head_next}, which reaches x_{last}.

Case $x_{head} = x_{last}$. This corresponds to the queue being empty, which we derive below.

As x_{last} is the last node, we have that out $x_{head} \mapsto$ None, hence the load resolves to None. Using the abstract points-to predicates from the invariant together with 4.8, 4.9, and lemma 12 we get $x_{head} \rightsquigarrow x'_{head}$ and $x_{head} \rightsquigarrow x_{tail'}$. We can now apply lemma 10 three times to conclude $x_{head} = x'_{head} = x_{tail'} = x_{tail}$. Since $x_{head'}$ points to None, then xs_{queue} has to be empty (if it wasn't we could deduce that $x_{head'}$ pointed to a node). This also implies that abstract state of the queue xs_v, is empty, $xs_v = []$.

Because the load resolved to None, then the variable *next* in the code will be None, and since we are in the case where the consistency check passes, and since we have derived that $x_{head} = x_{tail}$, we already know now that dequeue will return None. In other words, this is a linearisation point.

Since $xs_v = []$, then our abstract state predicate from the invariant states $Q_\gamma \Mapsto_\bullet []$. We thus instantiate the viewshift with $[]$, and supply the P from the pre-condition. As $xs_v = []$ we can conclude that the first disjunct must be true (the second contains a contradiction), so we get Q None and $Q_\gamma \Mapsto_\bullet []$. As we haven't changed any resources, we can close the invariant again.

We reach the consistency check on line 7. By 4.6, we know that $!(\mathsf{fst}(!\,Q))$ steps to $!(\ell_{head})$, but to resolve the prophecy, we must first show what $!(\ell_{head})$ evaluates to. This resource is inside the invariant so we open it. We get that $\ell_{head} \mapsto$ in x''_{head} for some node x''_{head}. We close the invariant and resolve the prophecy: $!(\mathsf{fst}(!\,Q))$ evaluated to in x''_{head}. In other words, $v_p =$ in x''_{head}, and therefore in $x_{head} =$ in x''_{head}. Since the remaining if statement on line 7 compares in x_{head} to in x''_{head}, we know that we will take the 'then' branch, so we step to line 8. Since $x_{head} = x_{tail}$ and *next* was set to None, we step to line 10 which returns None. The post-condition thus requires us to prove Q None, which we already have.

Case out $x_{head} \mapsto^{\square}$ in $x_{head\_next} * x_{head\_next} \rightsquigarrow x_{last}$. This means that the queue is not empty, and there is an element to be dequeued: the value in x_{head_next}. The load resolves to in x_{head_next}, and the program variable *next* is set to this. Using lemmas 8 and 13 with

4.15 we get

$$\text{in } x_{head\_next} \mapsto^{\square} (\text{val } x_{head\_next}, \text{out } x_{head\_next}) \tag{4.16}$$

$$x_{head\_next} \dashrightarrow Q_{\gamma}.\gamma_{Last} \tag{4.17}$$

We close the invariant and step to the consistency check on line 7. We handle this similarly to the previous case, and conclude that the consistency check succeeds and we take the 'then' branch to line 8. This line ensures that the dequeue will not make the tail node lag behind the head node. We can simply consider both cases of the check.
The case where the 'if' succeeds takes us to the CAS on line 12, which attempts to swing the tail, and try dequeueing again. We handle the CAS with our swing-tail lemma (lemma 16), and the recursive call by the induction hypothesis.
If the 'if' fails, then the tail node will not lag behind as a result of the dequeue, so we step to the else block on line 13 which attempts to dequeue. We first read the value out of x_{head_next} on line 14. Next, we attempt to swing the head pointer on line 15. The rule for CAS demands a points-to predicate for ℓ_{head}, so we open the invariant which gives us fresh nodes x'_{head}, x'_{tail}, and x'_{last}, so that $\ell_{head} \mapsto \text{in } x'_{head}$. The success of the CAS depends on whether in x'_{head} equals in x_{head}. If they aren't equal, the CAS fails and nothing is updated. We can thus close the invariant and we step to the recursive call on line 17 and apply the induction hypothesis. So for the remainder, we assume they are equal and the CAS succeeds. Since the CAS moved the head pointer to x_{head_next}, the dequeue will succeed, so this is a linearisation point.
Using the abstract points-to and reachability propositions from the invariant together with lemmas 12 and 8, we may deduce that x'_{head} is a node. As we already know that x_{head} is a node (from 4.7), we get that $x_{head} = x'_{head}$ from lemma 19. From the invariant (specifically $\text{isLL}\, xs$) we know that x'_{head} points to the first element of xs_{queue}. But since $x_{head} = x'_{head}$, then this element must be our x_{head_next}. In other words, $xs_{queue} = xs'_{queue} +\!\!+ [x_{head\_next}]$, for some xs'_{queue}. This means that, when we apply the viewshift (giving up p and $Q_{\gamma} \Mapsto_{\bullet} xs_v$ as usual), only the second case of the resulting disjunct is possible: xs_v cannot be empty as xs_{queue} isn't. We therefore get that there are some xs'_v and x_v so that

$$xs_v = xs'_v +\!\!+ [x_v] \tag{4.18}$$

$$Q_{\gamma} \Mapsto_{\bullet} xs'_v \tag{4.19}$$

$$Q\ (\text{Some}\, x_v) \tag{4.20}$$

Since xs_v is reflected in xs_{queue} (according to the invariant), we may additionally conclude that xs'_v is reflected in xs'_{queue} and $\text{Some}\, x_v = \text{val } x_{head\_next}$.

To close the invariant, we must update some of our resources. Since we now have $\ell_{head} \mapsto \text{in } x_{head\_next}$, we must pick x_{head_next} for the head node. But currently $Q_\gamma.\gamma_{Head} \rightarrowtail x_{head}$. So we use lemma 14 to advance the pointer, and we get $Q_\gamma.\gamma_{Head} \rightarrowtail x_{head\_next}$.

We are now required to show that $x_{head\_next} \dashrightarrow Q_\gamma.\gamma_{Tail}$. Since $x_{head} \rightsquigarrow x_{tail}$, and $x_{head} \neq x_{tail}$, then it must be the case that $x_{head\_next} \rightsquigarrow x_{tail}$. Lemma 12 with 4.13 now tells us that x_{tail} can reach the current tail node, x'_{tail}. By transitivity (lemma 7), we get $x_{head\_next} \rightsquigarrow x'_{tail}$, and hence by lemma 13, we get the desired $x_{head\_next} \dashrightarrow Q_\gamma.\gamma_{Tail}$.

We now own all the resources required to close the invariant. As the CAS succeed, we step to line 16 which simply returns val x_{head_next}. Thus, we must prove the post-condition Q (val x_{head_next}). We can do this since we still own 4.20 and we deduced that Some x_v = val x_{head_next}.

Case in $x_{head} \neq v_p$. We step to line 6 which finds out what x_{head} points to. To access the relevant points-to predicate, open the invariant. Importantly, we get that there is some last node of the linked list, x_{last}, with $Q_\gamma.\gamma_{Last} \rightarrowtail x_{last}$. By combining this with 4.10 and lemma 12 we deduce that $x_{head} \rightsquigarrow x_{last}$ which means that either x_{head} *is* the last node, and hence out $x_{head} \mapsto$ None, or there is some other node x_{head_next} and out $x_{head} \mapsto^{\square} x_{head\_next}$. This shows that the load is safe. The actual value that he load resolves to is unimportant, as it won't be used in this case. We thus perform the load, close the invariant and step to line 7.

By 4.6, we know that $!(\text{fst}(!\,Q))$ steps to $!(\ell_{head})$, so to resolve the prophecy, we show what $!(\ell_{head})$ evaluates to. We open the invariant, which gives us that $\ell_{head} \mapsto \text{in } x'_{head}$ for some node x'_{head}. We close the invariant again, and resolve the prophecy: $!(\text{fst}(!\,Q))$ evaluated to in x'_{head}. That is, $v_p = \text{in } x'_{head}$, and therefore in $x_{head} \neq \text{in } x'_{head}$. We hence take the 'else' branch and step to line 18. This consists of a recursive call to the loop function, hence we are done by the induction hypothesis.

□

4.5 Discussion

▶Mention that one can remove the consistency check and hence also the prophecy◀

Chapter 5

Conclusion and Future work

▶conclude on the problem statement from the introduction◀

▶Mention the possibility of simplifying queue invariant for lock-free by removing isLL (and adding x_last -> None)◀

Bibliography

[1] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In James E. Burns and Yoram Moses, editors, *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, USA, May 23-26, 1996*, pages 267–275. ACM, 1996.

[2] Simon Friis Vindum and Lars Birkedal. Contextual refinement of the michael-scott queue (proof pearl). In Catalin Hritcu and Andrei Popescu, editors, *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*, pages 76–90. ACM, 2021.

Appendix A

The Technical Details

Lemma 19 (Node equality)**.**

$$\begin{aligned}
&\forall x, y. \\
&\ \mathrm{in}\ \ x = \mathrm{in}\ \ y \mathrel{-\!\!*} \\
&\ \mathrm{in}\ \ x \mapsto^{\square} (\mathrm{val}\ \ x, \mathrm{out}\ \ x) \mathrel{-\!\!*} \\
&\ \mathrm{in}\ \ y \mapsto^{\square} (\mathrm{val}\ \ y, \mathrm{out}\ \ y) \mathrel{-\!\!*} \\
&x = y
\end{aligned}$$

►. . . ◄