
TITLE HERE

Mathias Pedersen, 201808137

Master's Thesis, Computer Science

March 2024

Advisor: Amin Timany



AARHUS
UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

Abstract

► in English... ◄

Resumé

► in Danish... ◄

Acknowledgments



Mathias Pedersen
Aarhus, March 2024.

Contents

Abstract	iii
Resumé	v
Acknowledgments	vii
1 Introduction	1
2 Preliminaries	3
3 The Two-Lock Michael Scott Queue	5
3.1 Preliminaries	5
3.2 implementation	6
3.2.1 initialise	6
3.2.2 enqueue	6
3.2.3 dequeue	6
3.3 Sequential Specification	8
3.4 Proving the Sequential Specification	9
3.4.1 The isqueue predicate	9
3.4.2 Proof outline	10
3.5 Concurrent Specification	11
4 Conclusion	13
Bibliography	15
A The Technical Details	17

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

►Description of HeapLang, Iris, Verified in Coq (Weakest precondition vs Hoare triples)◄

Chapter 3

The Two-Lock Michael Scott Queue

I present here 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 $(\ell_{i_in}, v_i, \ell_{i_out})$. The location ℓ_{i_in} points to the pair (v_i, ℓ_{i_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 $(\ell_{i_in}, v_i, \ell_{i_out})$, we mean that $\ell \mapsto \ell_{i_in}$. Hence, if we have two adjacent nodes $(\ell_{i_in}, v_i, \ell_{i_out})$, $(\ell_{i+1_in}, v_{i+1}, \ell_{i+1_out})$ in the linked list, then we have the following structure: $\ell_{i_in} \mapsto (v_i, \ell_{i_out})$, $\ell_{i_out} \mapsto \ell_{i+1_in}$, and $\ell_{i+1_in} \mapsto v_{i+1}, \ell_{i+1_out}$.

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 which I now present in turn.

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 they will always point to (v_i, ℓ_{i_out}) . We shall use the notation $\ell \mapsto \Box v$ (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.

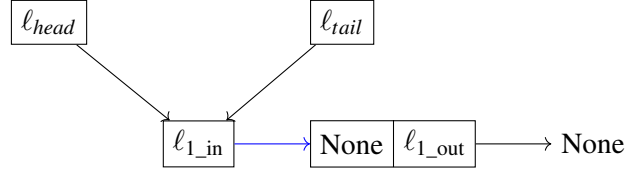
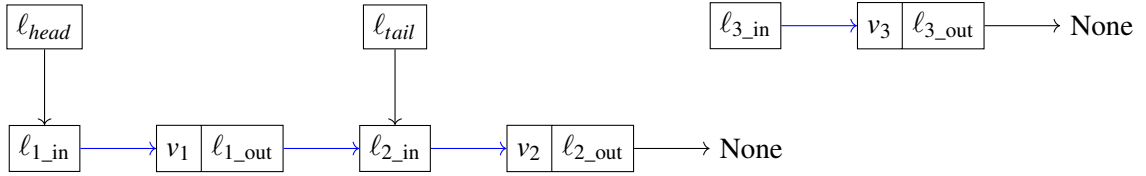
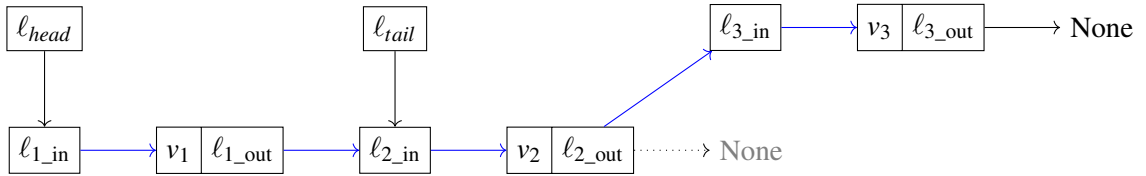


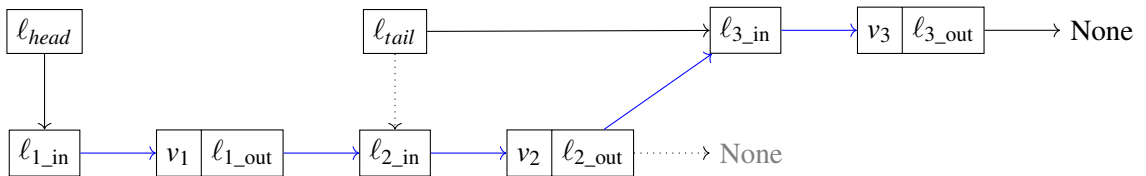
Figure 3.1: Queue after initialisation



(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.

let 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))

```

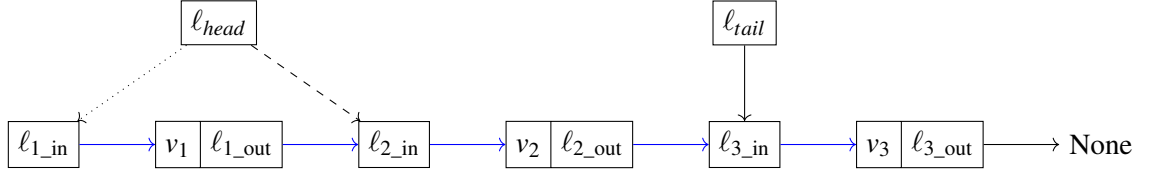


Figure 3.3: Dequeueing an element (v_2) from a queue with two elements (v_2, v_3). The dotted line represents the state before the dequeue, and the dashed line is the state after dequeuing.

```

let enqueue  $Q$  value :=
  let node = ref ((Some value, ref (None))) in
  acquire(snd(snd(! $Q$ )));
  snd(!(!snd(fst(! $Q$ ))))  $\leftarrow$  node;
  snd(fst(! $Q$ ))  $\leftarrow$  node;
  release(snd(snd(! $Q$ )))

```

```

let 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
    fst(fst(! $Q$ ))  $\leftarrow$  new_head;
    release(fst(snd(! $Q$ )));
    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 allow 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 dequeuing 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 "*Qghostnames*" 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, element of *Qghostnames* will simply be pairs. For an element $Q_\gamma \in Qghostnames$, 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:

$$\begin{aligned} & \exists \text{is_queue} : Val \rightarrow List\ Val \rightarrow Qghostnames \rightarrow Prop. \\ & \{ \text{True} \} \text{ initialize}() \{ v. \exists Q_\gamma, \text{is_queue } v \ [] \ Q_\gamma \} \\ & \wedge \quad \forall q, v, xs_v, Q_\gamma. \{ \text{is_queue } q \ xs_v \ Q_\gamma \} \text{ enqueue } q \ v \{ v. \text{is_queue } q \ (v :: xs_v) \ Q_\gamma \} \\ & \wedge \quad \forall q, xs_v, Q_\gamma. \{ \text{is_queue } q \ xs_v \ Q_\gamma \} \text{ dequeue } q \left\{ v. \begin{array}{l} (xs_v = [] * v = \text{None} * \text{is_queue } q \ xs_v \ Q_\gamma) \vee \\ (\exists x_v, xs'_v. xs_v = xs'_v ++ [x_v] * v = \text{Some } x_v * \text{is_queue } q \ xs'_v \ Q_\gamma) \end{array} \right\} \end{aligned}$$

The predicate $\text{is_queue } q \ xs_v \ Q_\gamma$ captures that the value 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 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 Predicate

To prove the specification we must give a specific is_queue 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.

$$\begin{aligned}
\text{is_queue } q \text{ } xs_v \text{ } Q_\gamma &= \exists l_{\text{queue}}, \text{head}, \text{tail} \in \text{Loc}. \exists H_{\text{lock}}, T_{\text{lock}} \in \text{Val}. \\
q &= l_{\text{queue}} * l_{\text{queue}} \mapsto \square((\text{head}, \text{tail}), (H_{\text{lock}}, T_{\text{lock}})) * \\
&\exists xs_{\text{queue}} \in \text{List}(\text{Loc} \times \text{Val} \times \text{Loc}). \exists x_{\text{head}}, x_{\text{tail}} \in (\text{Loc} \times \text{Val} \times \text{Loc}). \\
&\text{proj_val } xs_{\text{queue}} = \text{wrap_some } xs_v * \\
&\text{isLL}(xs_{\text{queue}} ++ [x_{\text{head}}]) * \\
&\text{head} \mapsto (\text{in } x_{\text{head}}) * \\
&\text{tail} \mapsto (\text{in } x_{\text{tail}}) * \text{isLast } x_{\text{tail}} (xs_{\text{queue}} ++ [x_{\text{head}}]) * \\
&\text{isLock } Q_\gamma \cdot \gamma_{H_{\text{lock}}} H_{\text{lock}} \text{ True} * \\
&\text{isLock } Q_\gamma \cdot \gamma_{T_{\text{lock}}} T_{\text{lock}} \text{ True}.
\end{aligned}$$

This `is_queue` predicate states that the value q is a location, which always 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 (represented by xs_{queue}), by stating that if you strip away the locations connecting the nodes and remove the `Some` around the values, then you get the abstract state xs_v . 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. 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 3.4.1 (Linked List Chain Predicate)

$$\begin{aligned}
\text{isLL_chain } [] &\equiv \text{True} \\
\text{isLL_chain } [x] &\equiv \text{in } x \mapsto \square(\text{val } x, \text{out } x) \\
\text{isLL_chain } x :: x' :: xs &\equiv \text{in } x \mapsto \square(\text{val } x, \text{out } x) * \text{out } x' \mapsto \square \text{in } x * \text{isLL_chain } x' :: xs
\end{aligned}$$

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

Definition 3.4.2 (Linked List Predicate)

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

3.4.2 Proof outline

►give proof outline◄

3.5 Concurrent Specification

►update to reflect changes in Coq◄

$$\begin{aligned}
& \exists \text{is_queue} : \text{Val} \mapsto \text{Qnames} \mapsto \text{Prop}. \\
& \quad \forall v, Q_\gamma. \text{is_queue } v \ Q_\gamma \implies \Box \text{is_queue } v \ Q_\gamma \\
& \quad \wedge \quad \{\text{True}\} \text{ initialize } () \{v. \exists Q_\gamma. \text{is_queue } v \ Q_\gamma\} \\
& \quad \wedge \quad \forall q, v, Q_\gamma. \{\text{is_queue } v \ Q_\gamma\} \text{ enqueue } q \ v \ \{v. \text{True}\} \\
& \quad \wedge \quad \forall q, v, Q_\gamma. \{\text{is_queue } v \ Q_\gamma\} \text{ dequeue } q \ \{v. \text{True}\}
\end{aligned}$$

Key insights (concurrent)

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**)

Insight 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 3.5). 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} .

With these insights, we can now define the queue invariant.

Definition 3.5.1 (Two-Lock M&S-Queue Invariant) ►Write it in◄

The specification for the two-lock Michael Scott Queue *can* be proven using the queue invariant 3.5.1. However, during the proof, a simpler (but arguably less intuitive) queue invariant was discovered. This simpler invariant is equivalent to 3.5.1 and has the added benefit of being easier to work with in the proofs. Thus, we shall be using this version of the queue invariant during the proofs.

Definition 3.5.2 (Simplified Two-Lock M&S-Queue Invariant) ►Write it in◄

Discussing the need for xs_{old}

► **Update to reflect changes in Coq** ◀ As mentioned in the insights, 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 the head to the new node, we get that the entire queue is xs . After performing the write, 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"). Because of this, we can supply the same predicate concerning the *tail* (the or) that we opened the queue with, since these only mention xs , which remains the same.

Had we not used an xs_{old} and essentially just "forgotten" old nodes in the queue, we couldn't have done this. Say that we defined xs as $xs = x_{head} :: xs_{rest}$ instead. Then, once we have to close the invariant, we cannot supply the xs , which we got when we opened the invariant. Our only choice (due to the fact that *head* must point to $x_{n_{head}}$) is to close the invariant with $xs' = xs_{rest} = x_{n_{head}} :: xs''_{n_{rest}}$. However, clearly $xs' \neq xs$, so we cannot supply the same predicate concerning the *tail* (the or) that we got when opening the invariant, since this predicate talks about xs , not xs' . Now, if we opened the invariant in the Dequeue case, then we could assert that $lastxs = lastxs'$, and hence still be able close the invariant. However, if we opened the invariant in the Both case, then we would need to assert that $2lastxs = 2lastxs'$. This is however not provable, since it might be the case that $xs''_{n_{rest}}$ is empty, and hence $2lastxs'$ is *None*, whereas $2lastxs = x_{n_{head}}$.

Chapter 4

Conclusion

►conclude on the problem statement from the introduction◄

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

