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

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 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 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 $(v_i, \ell_{i\_\mathrm{out}})$. 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.

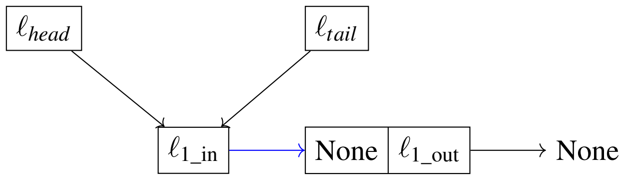
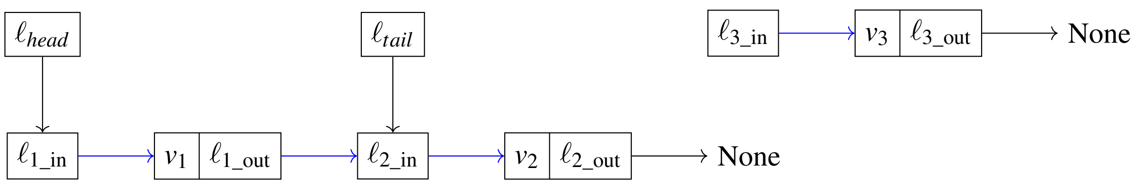
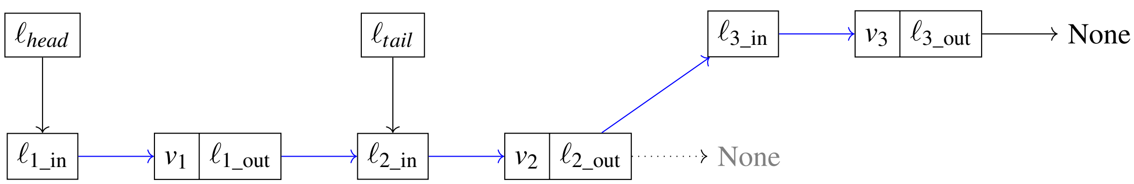


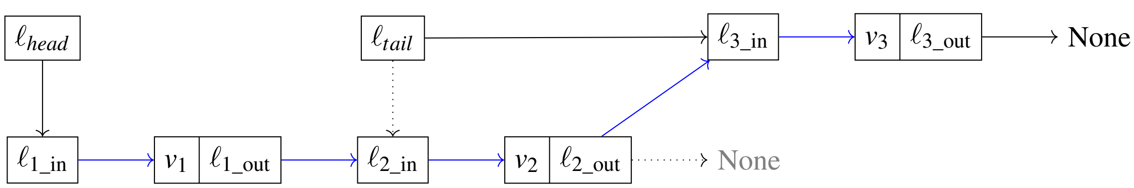
Figure 3.1: Queue after initialisation



(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.

$$\mathsf{let}\ \mathrm{initialize} :=$$
$$\mathsf{let}\ node = \mathsf{ref}\ ((\mathrm{None}, \mathsf{ref}\ (\mathrm{None})))\ \mathsf{in}$$
$$\mathsf{let}\ H\_lock = newlock()\ \mathsf{in}$$
$$\mathsf{let}\ T\_lock = newlock()\ \mathsf{in}$$
$$\mathsf{ref}\ ((\mathsf{ref}\ (node), \mathsf{ref}\ (node)), (H\_lock, T\_lock))$$

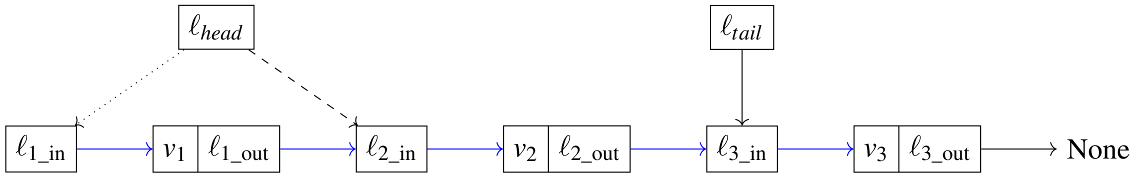


Figure 3.3: Dequeueing an element (v_2) from a queue with two elements (v_2, v_3). The dotted line represents the state before the dequeue, and the dashed line is the state after dequeuing.

$$
\begin{aligned}
&\mathsf{let}\,\text{enqueue}\ Q\ \mathit{value} :=\\
&\qquad \mathsf{let}\ \mathit{node} = \mathsf{ref}\ ((\text{Some}\,\mathit{value}, \mathsf{ref}\ (\text{None})))\ \mathsf{in}\\
&\qquad \text{acquire}(\mathsf{snd}(\mathsf{snd}(!\,Q)));\\
&\qquad \mathsf{snd}(!(!(\mathsf{snd}(\mathsf{fst}(!\,Q))))) \leftarrow \mathit{node};\\
&\qquad \mathsf{snd}(\mathsf{fst}(!\,Q)) \leftarrow \mathit{node};\\
&\qquad \text{release}(\mathsf{snd}(\mathsf{snd}(!\,Q)))
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{let}\,\text{dequeue}\ Q :=\\
&\qquad \text{acquire}(\mathsf{fst}(\mathsf{snd}(!\,Q)));\\
&\qquad \mathsf{let}\ \mathit{node} = !(\mathsf{fst}(\mathsf{fst}(!\,Q)))\ \mathsf{in}\\
&\qquad \mathsf{let}\ \mathit{new\_head} = !(\mathsf{snd}(!\,\mathit{node}))\ \mathsf{in}\\
&\qquad \mathsf{if}\ \mathit{new\_head} = \text{None}\ \mathsf{then}\\
&\qquad\quad \text{release}(\mathsf{fst}(\mathsf{snd}(!\,Q)));\\
&\qquad\quad \text{None}\\
&\qquad \mathsf{else}\\
&\qquad\quad \mathsf{let}\ \mathit{value} = \mathsf{fst}(!\,\mathit{new\_head})\ \mathsf{in}\\
&\qquad\quad \mathsf{fst}(\mathsf{fst}(!\,Q)) \leftarrow \mathit{new\_head};\\
&\qquad\quad \text{release}(\mathsf{fst}(\mathsf{snd}(!\,Q)));\\
&\qquad\quad \mathit{value}
\end{aligned}
$$

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. $xs_v : \mathit{List\ Val}$. 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 "*Qgnames*" 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 *Qgnames* will simply be pairs. For an element $Q_\gamma \in Qgnames$, 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 head lock.

The sequential specification we wish to prove is the following:

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

The predicate is_queue $v_q\ xs_v\ Q_\gamma$ 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 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.

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

This is_queue predicate states that the value v_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)

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

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

Definition 3.4.2 (Linked List Predicate)

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

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.

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 with the obtained resources. To begin with, we step through the lines creating the first node, giving us locations $\ell_{1\_\text{in}}, \ell_{1\_\text{out}}$ with $\ell_{1\_\text{out}} \mapsto$ 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 $\ell_{1\_\text{in}} \mapsto \square(\text{None}, \ell_{1\_\text{out}})$. 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, $\gamma_{Tlock}, \gamma_{Tlock}$, which we will collect in a *Qgnames* pair, Q_γ. Next, we step through the allocations of the head, tail, and queue, which gives us locations $\ell_{head}, \ell_{tail}, \ell_{queue}$, such that both ℓ_{head} and ℓ_{tail} point to node 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: $\exists Q_\gamma.\, \text{is\_queue}\ \ell_{queue}\ Q_\gamma$. 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 1 for xs_{head} and xs_{tail}.

Enqueue

►add line numbers to code, and refer to them in proof◄ For enqueue, we get in our pre-condition is_queue $v_q\ xs_v\ Q_\gamma$, and we wish to that, if we run enqueue $v_q\ v$, then we will get is_queue $v_q\ (v :: xs_v)\ Q_\gamma$. The proposition is_queue $v_q\ xs_v\ Q_\gamma$ 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 $x_n ew$ 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 $x_n ew$. The resources needed to do this are all described in is_queue $v_q\ xs_v\ Q_\gamma$. 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 ($xs_{queue} ++ [x_{head}]$). 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 $v_q\ (v :: xs_v)\ Q_\gamma$. 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 $xs_n ew$ cons'ed to it, and for x_{tail}, we chose the new tail node: x_{new}. With these choices, proving is_queue $v_q\ (v :: xs_v)\ Q_\gamma$ is fairly straightforward.

Dequeue

For dequeue v_q, our precondition is is_queue $v_q\ xs_v\ Q_\gamma$, 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 $v_q\ xs_v\ Q_\gamma$. The is_queue 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 $v_q\ xs_v\ Q_\gamma$ 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. $xs_{queue} = xs'_{queue} ++ [x_{head\_next}]$. 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 $v_q\ xs_v\ Q_\gamma$.
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 $v_q\ xs'_v\ Q_\gamma$, 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 is_queue 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.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\exists \text{is\_queue} : (\mathit{Val} \to \mathsf{Prop}) \to \mathit{Val} \to \mathit{Qgnames} \to \mathsf{Prop}.} \\
\multicolumn{2}{l}{\forall \Phi : \mathit{Val} \to \mathsf{Prop}.} \\
 & \forall v_q, Q_\gamma.\, \text{is\_queue } \Phi\ v_q\ Q_\gamma \implies \Box \text{is\_queue } \Phi\ v_q\ Q_\gamma \\
\wedge & \{\mathsf{True}\}\ \mathsf{initialize}()\ \{v_q.\exists Q_\gamma, \text{is\_queue } \Phi\ v_q\ Q_\gamma\} \\
\wedge & \forall v_q, v, Q_\gamma.\{\text{is\_queue } \Phi\ v_q\ Q_\gamma * \Phi\ v\}\ \text{enqueue}\ v_q\ v\ \{v.\mathsf{True}\} \\
\wedge & \forall v_q, Q_\gamma.\{\text{is\_queue } \Phi\ v_q\ Q_\gamma\}\ \text{dequeue}\ v_q\ \{v.v = \text{None} \vee (\exists x_v, v = \text{Some}\, x_v * \Phi\ x_v)\}
\end{array}
$$

3.6 Proving the Concurrent Specification

3.6.1 The is_queue 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 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 Invariant)

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

In contrast to the sequential specification, the abstract state is now existentially quantified, hence the exact contents of the queue are not tracked. But the concrete state of the queue is still reflected in the abstract state through a value projection and wrapping of values. 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 proposition $All\ xs_v\ \Phi$ 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 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 $\ell_{tail} \mapsto 1/2 x_{tail}$ from an earlier access, and it will obtain the existence of some new x'_{tail}, such that $\ell_{tail} \mapsto 1/2 x'_{tail}$. Combining the two points-to predicates allows us to conclude that $x_{tail} = x'_{tail}$. In this way, we can match up variables from earlier access 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, or the new node has been added, but the tail pointer is still pointing to the previous tail node.
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: $\mathrm{Ex}()$. 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 $\mathrm{Ex}()$. 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 TokE Q_γ 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.

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

$$\begin{aligned}\text{is\_queue } v_q\ xs_v\ Q_\gamma =& \exists \ell_{queue}, \ell_{head}, \ell_{tail} \in Loc. \exists H_{lock}, T_{lock} \in Val. \\ & v_q = \ell_{queue} * \ell_{queue} \mapsto \square((\ell_{head}, \ell_{tail}), (H_{lock}, T_{lock})) * \\ & \exists \iota. \boxed{queue\_invariant \Phi \ell_{head} \ell_{tail} Q_\gamma}^{\iota} * \\ & isLock\ Q_\gamma.\gamma_{Hlock}\ H_{lock}\ (\text{TokD } Q_\gamma) * \\ & isLock\ Q_\gamma.\gamma_{Tlock}\ T_{lock}\ (\text{TokE } Q_\gamma).\end{aligned}$$

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 predicate.

3.6.2 Proof outline

Firstly, we must show that is_queue 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 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 $\ell_{tail} \mapsto 1/2\,\text{in}\ x_{tail}$. 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 $\ell_{tail} \mapsto 1/2\,\text{in}\ x'_{tail}$. However, since we kept $\ell_{tail} \mapsto 1/2\,\text{in}\ x_{tail}$, 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 $\ell_{tail} \mapsto 1/2 x''_{tail}$, 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 $\ell_{head} \mapsto 1/2 x'_{head}$. 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. We are now left with the return value: None. We change the post-condition to prove the left disjunct. We finish the proof using Ht-ret.

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 $\ell_{head} \mapsto 1/2 x''_{head}$. 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, framing away Φx_v and concluding with Ht-ret.

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

3.7.1 Proof outline

The proofs are largely similar to the concurrent spec. For initialise, we must additionally allocate the ghost resource **►auth◄**$([]) \cdot$**►frac◄**$([])$. We can only do this if the element is valid, but this follows by the definitions of the resource algebras. 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 **►auth◄**(xs_v) to **►auth◄**$(v :: xs_v)$. 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 **►auth◄**(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 **►auth◄**$(v :: xs_v)$. However, since we have **►auth◄**(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.

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

►...◄