Verification of the Blocking and Non-Blocking Michael-Scott Queue Algorithms

Mathias Pedersen, 201808137

Master's Thesis, Computer Science

May 2024

Advisor: Amin Timany

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

Ensuring correctness of programs is of ever-growing importance as software increasingly handles safety-critical functions. The surge of concurrent programs makes this task significantly more challenging. This thesis focuses on queue algorithms which are commonly used in a concurrent setting.
To reason formally about queues, we use the Iris program logic. After giving a brief introduction to the logic, we use it to express three queue specifications. The most general of these is the "Hocap-style" specification, which supports concurrent clients and allows tracking of queue contents. We demonstrate that this specification implies the other, simpler specifications.
The project then examines two queue algorithms: the blocking and non-blocking Michael-Scott Queues. We give implementations for these algorithms in the programming language HeapLang. Both implementations are proven to satisfy the Hocap-style specification. The blocking Michael-Scott Queue is verified using an invariant that tracks the contents of the queue and specifies the possible states of the queue data structure. For the non-blocking Michael-Scott Queue, a notion of reachability, borrowed from existing work, is incorporated into its invariant to meet the Hocap-style specification. All proofs developed in this project have been mechanised in the Coq proof assistant.

Resumé

I forbindelse med at software overtager flere sikkerheds-kritiske opgaver stiger vigtigheden af verificering af software. Den øgede popularitet af "parallelprogrammering" gør denne verificerings-opgave markant mere udfordrende. Dette speciale fokuser på kø-algoritmer, som ofte bruges i parallelprogrammering.
Vi bruger program logikken Iris til at ræsonnere formelt omkring køer. Efter en kort introduktion til logikken bruger vi den til at udtrykke tre specifikationer for køer. Den mest generelle af disse er den såkaldte "Hocap-style" specifikation, som tillader køen at blive brugt i parallelle sammenhænge, samt understøtter at klienter kan holde styr på køens indhold. Vi demonstrerer at køer, der overholder Hocap-style specifikationen, også overholder de to andre specifikationer.
Projektet udforsker derefter to specifikke køer: den blokerende og ikke-blokerende Michael-Scott kø. Vi give implementeringer af køerne i programmeringssproget HeapLang. Det vises, at begge implementeringer overholder Hocap-style specifikationen. Verifiseringen af den blokerende Michael-Scott kø fungerer ved brugen af en invariant, som holder styr på køens indhold, og specificerer de mulige tilstande som køren kan være i. Invarianten for den ikke-blokerende Michael-Scott kø benytter ideen om "reachability" fra tidligere arbejde. Alle beviser fremstillet i projektet er blevet maskine-verificeret i bevis assistenten Coq.

Acknowledgments

I would like to thank my advisor, Amin Timany, for giving great guidance and feedback and for being a great teacher in both this and previous projects.
A special thanks to my family who has given me support and encouragement throughout my studies.

Mathias Pedersen
Aarhus, May 2024.

Contents

Chapter 1

Introduction

Correctness of software is of ever-growing importance as more and more safety-critical functions are entrusted to computers. As such, being able to verify the correctness of a piece of software has become highly desirable. With the surge of multiprocessor computing, the complexity of software systems has grown significantly. Reasoning about the correctness of concurrent programs is particularly tricky as one must reason about all possible interactions between participating threads.

A verification technique that gives very strong correctness guarantees is that of using a *program logic*. The idea is to set up a formal system that captures the semantics of a programming language, which one can then use to derive (i.e. prove) formal descriptions of how specific expressions behave: a specification of the program. That is, the logic allows us to write formal program specifications and prove them. One such program logic is *Iris*, a concurrent separation logic that supports reasoning about concurrent programs and how the resources these programs operate on are used by participating threads.

This project focuses on *queues*. Using the aforementioned program logic, we give specifications for queue algorithms in general. These specifications vary in generality and complexity, reflecting the difficulties of reasoning about concurrent programs. The most general queue specification we develop is a so-called *Hocap style* specification, which supports concurrent clients, and allows them to track the contents of the queue.

In this project, we study two concrete queue implementations, namely the blocking and non-blocking Michael-Scott Queues (M&S Queues for short). These queues demonstrate two of the most common synchronisation mechanisms: locks, which are blocking, and atomic instructions such as compare-and-swap, which are non-blocking.

Previous work [Vindum and Birkedal, 2021] has shown that the non-blocking M&S Queue contextually refines a coarse-grained queue. However, this result does not allow clients of the queue to formally verify their own specifications using the logic.
As for the blocking M&S Queue, no prior correctness results are known to this author. The original presentation of the queue [Michael and Scott, 1996] gave an argument for its correctness, however, we demonstrate that the argument relies on an incorrect assumption.

This project makes the following contributions.

- Giving three different specifications for queues at different levels of generality.
- Proving that both M&S Queues meet these specifications.
- Demonstrating that the Hocap-style specification implies the other two specifications.
- Mechanising everything presented in this report in the Coq proof assistant.

The structure of the report is as follows. We begin in chapter 2 with an exposition of the programming language used in the project, as well as the program logic, Iris. Following this, in chapter 3 we give the three queue specifications and discuss their capabilities. We further prove that the Hocap style specification can derive the other two specifications. Chapter 4 discusses the Two-Lock M&S Queue and gives an implementation for it in our chosen programming language. We then proceed to prove that the implementation satisfies the three specifications in chapter 5. In a similar way, chapter 6 gives our implementation of the Lock-Free M&S Queue, and chapter 7 shows that it satisfies the Hocap style specification.

Chapter 2

Preliminaries

This chapter covers some of the background topics that are required to understand the main parts of the project. Specifically, section 2.1 covers the basics of the language we use to implement the M&S Queues. Section 2.2 gives an overview of the program logic we use to reason about the queues. Finally, section 2.3 introduces the files containing the mechanisations of the work presented in this report.

2.1 HeapLang

This project uses "HeapLang" as the implementation language. The main reason for this choice is that HeapLang is very well supported by the program logic Iris, which we introduce in the next section. However, HeapLang is still a quite suitable language for this project, as it has features to support faithful implementations of the M&S Queues. In particular, HeapLang is an untyped ML-style language and notably supports recursion, references, and concurrency. References are handled with a *heap*, hence the name of the language. Only reference to values can be created. The instruction $\mathsf{ref}\,(v)$ will allocate a spot on the heap containing v and return the location, ℓ, of the value. This location can then later be read using $!\,\ell$, or updated using $\ell \leftarrow v'$.

Concurrency is supported by the $\mathsf{fork}\ \{e\}$ instruction. This instruction creates a new thread which executes e. Multiple threads can communicate through the heap, and to allow for synchronisation, HeapLang supports a compare-and-swap instruction. The instruction $\mathsf{CAS}\ \ell\ v\ v'$ atomically reads ℓ, compares the contents with v, and if similar, stores v' at ℓ. If ℓ does not contain v, no change occurs.

For the basic constructs, HeapLang supports basic arithmetic on integers, comparisons, conditionals, pairs and projections, and injections and pattern matching. Many of the other usual constructions expected of similar languages are achieve with syntactic sugar. For instance, we define the let construction as $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \triangleq (\lambda x.e_2)\ e_1$. Similarly, we can support sequences of instructions by defining $e_1; e_2 \triangleq (\lambda x.e_2)\ e_1$, but where x is fresh.

The full specification of the language can be found in Birkedal and Bizjak [2017] (section 2 at time of writing).

2.2 The Iris Program Logic Framework

In this section, we give a brief introduction to Iris – the logic we use to reason about the M&S Queues. Iris is quite expressive and supports a myriad of features and derived rules, many of which have been utilised in this project. As such, it will be impossible to cover all facets of Iris in detail, so we limit ourselves to give an overview of the main aspects of Iris. If the reader wishes a more thorough introduction, or wants to see further details of the topics covered, please consult the Iris Lecture Notes [Birkedal and Bizjak, 2017].

2.2.1 Fundamentals of Iris

Briefly put, Iris is a "Higher-Order Concurrent Separation Logic Framework". The framework part means that Iris is not tied to a single programming language; one may instantiate Iris with any programming

language one sees fit. The "concurrent" part implies that Iris supports reasoning about languages with concurrent features, such as HeapLang. In fact, HeapLang is a sort of "default" language which ships with Iris. As such, some of the rules we present in this section will be programming language independent, while others will adhere to the semantics of HeapLang.

A feature that makes Iris very powerful, is that it is a *higher order logic*. This essentially means that we may quantify over propositions and predicates. This feature of Iris is essential to this project – much of the work done relies on this capability.

As Iris is a separation logic, we can reason about ownership of *resources*. Iris's notion of resources is very general due to *resource algebras*, which we explain in section 2.2.4, but a simple example is that of pointers. One may "own" a particular pointer which allows one to manipulate it. We capture this with the "points-to" predicate; the proposition $\ell \mapsto v$ denotes ownership over location ℓ, with the fact that ℓ points-to v. It guarantees that no other threads can interact with the location. Ownership of $\ell \mapsto v$ can also be transferred to e.g. other threads, allowing them to interact with ℓ. In general, propositions in Iris describe the resources that one owns.

Iris has usual connectives such as $\wedge$, $\vee$, and $\Longrightarrow$, but with the addition of ownership, we additionally introduce *separating conjunction*, written as: $P * Q$, for propositions P and Q. The proposition $P * Q$ describes the resources in P *combined* with the resources in Q. Since ownership of a resource can be exclusive, it shouldn't be freely duplicable. Hence, if we own some resources and wish to prove $Q_1 * Q_2$, we must decide which resources we use to prove Q_1 and which resources to prove Q_2. This is captured by the $*\text{I}$ rule. This is the main difference between regular conjunction and separating conjunction. We additionally introduce the "wand" connective, written $-\!\!*$, which is similar to implication but works with separating conjunction instead. The introduction and elimination rules for wand ($-\!\!*\text{I}$ and $-\!\!*\text{E}$) are hence similar to those for implication, but it takes the ownership aspect into consideration.

$$*\text{I}\quad \frac{P_1 \vdash Q_1 \qquad P_2 \vdash Q_2}{P_1 * P_2 \vdash Q_1 * Q_2} \qquad -\!\!*\text{I}\quad \frac{R * P \vdash Q}{R \vdash P -\!\!* Q} \qquad -\!\!*\text{E}\quad \frac{R_1 \vdash P -\!\!* Q \qquad R_2 \vdash P}{R_1 * R_2 \vdash Q}$$

Ownership of resources does not have to be exclusive. For instance, if a location is never updated, then several threads should be allowed to have ownership over the points-to predicate for that location. To this end, Iris has the *persistently modality*, written $\Box\, P$ (read as "persistently P"), which states that the resources described by P are allowed to be duplicated (see rule PERSISTENTLY-DUP). Duplicability is an important capability; essential constructions in Iris rely on being duplicable as we explore in the next sections.

Propositions are persistent if they satisfy $P \vdash \Box\, P$. There are many rules which state how to derive and work with persistent propositions, but for the sake of brevity, we highlight here just two important rules: PERSISTENTLY-INTRO and PERSISTENTLY-KEEP. The first states that any proposition we prove from *only* persistent proposition are themselves persistent, and the second states that we can use resources to prove persistent propositions without "consuming" the resources.

$$\text{PERSISTENTLY-DUP}\quad \Box\, P \dashv\vdash \Box\, P * P \qquad \text{PERSISTENTLY-INTRO}\quad \frac{\Box\, P \vdash Q}{\Box\, P \vdash \Box\, Q} \qquad \text{PERSISTENTLY-KEEP}\quad \frac{P \vdash \Box\, Q}{P \vdash \Box\, Q * P}$$

2.2.2 Hoare Triples and Weakest Pre-Condition

The logic supports reasoning about programs via *Hoare triples*. A Hoare triple $\{P\}\, e\, \{v.\Phi\ v\}$ states that, if we own the resources described by P, then we may safely run e, and *if* the computation terminates with value v, then the predicate Φ holds of v. That is, Hoare triples only show partial correctness of programs.

Specifications for functions are usually written in terms of Hoare-triples – the pre-condition mentions which resources the function require and the post-condition states which resources the callee gets back, if the function returns. Hoare-triples are persistent (see rule PERSISTENTLY-HT), which allows clients to apply the same Hoare-triple for multiple invocations of the same function. This is sensible as the Hoare triple implies that the function only needs the resources described by the pre-condition in order to run safely; if we can get those resources multiple times, we can of course also run the function multiple times.

PERSISTENTLY-HT
$$\{P\}\, e\, \{\Phi\} \dashv\vdash \Box\, \{P\}\, e\, \{\Phi\}$$

HT-RET
$$\frac{w \text{ is a value}}{S \vdash \{\mathsf{True}\}\, w\, \{v.v = w\}}$$

HT-BIND
$$\frac{E \text{ is an eval. context} \qquad S \vdash \{P\}\, e\, \{v.\, Q\} \qquad S \vdash \forall v.\, \{Q\}\, E[v]\, \{w.\, R\}}{S \vdash \{P\}\, E[e]\, \{w.\, R\}}$$

HT-ALLOC
$$\frac{}{S \vdash \{\mathsf{True}\}\ \mathsf{ref}\ (u)\ \{v.\exists \ell.\, v = \ell \wedge \ell \hookrightarrow u\}}$$

HT-LOAD
$$\frac{}{S \vdash \{\ell \hookrightarrow u\}\ !\ell\ \{v.v = u \wedge \ell \hookrightarrow u\}}$$

HT-STORE
$$\frac{}{S \vdash \{\ell \hookrightarrow -\}\, \ell \leftarrow w\, \{v.v = () \wedge \ell \hookrightarrow w\}}$$

HT-BETA
$$\frac{S \vdash \{P\}\, e\, [v/x]\ \{u.Q\}}{S \vdash \{P\}\ (\lambda x.e)v\ \{u.Q\}}$$

HT-CSQ
$$\frac{S \text{ persistent} \qquad S \vdash P \Rightarrow P' \qquad S \vdash \{P'\}\, e\, \{v.\, Q'\} \qquad S \vdash \forall u.\, Q'[u/v] \Rightarrow Q[u/v]}{S \vdash \{P\}\, e\, \{v.\, Q\}}$$

HT-FRAME
$$\frac{S \vdash \{P\}\, e\, \{v.Q\}}{S \vdash \{P * R\}\, e\, \{v.Q * R\}}$$

Figure 2.1: Selected rules for Hoare triples.

The rules for Hoare triples depend on the language. Figure 2.1 shows some selected rules for the case of HeapLang. Some of the rules, such as HT-ALLOC, HT-LOAD, HT-STORE, and HT-BETA shows Hoare triples for basic constructs of the language. For example, HT-ALLOC states that it is always safe to create a new reference to some value, u, and the value returned from the creation is some location which points-to u.

Other rules such as HT-BIND, HT-CSQ, and HT-FRAME shows how to prove Hoare triples for more complex expressions. The bind rule allows us to "focus" on sub-expressions, as long as they are what will be executed next according to the semantics of the language. We must then prove a Hoare triple for the whole expression assuming that the sub-expression has reduced to some value. Note that the post-condition of the Hoare triple for the sub-expression becomes the pre-condition for Hoare triple of the whole expression.

The rule of consequence states that we can strengthen pre-conditions and weaken post-conditions. Indeed, if we can execute an expression with some resources, we can surely also execute it if we have *more* resources available. Similarly, we can also simply decide to mention *fewer* of the resources available after the function invocation.

The frame rule essentially states that resources which are not required to run the expression are not changed by it.

Another way to reason about programs in Iris is *weakest pre-condition*, written $\mathsf{wp}\, e\, \{v.\Phi\ v\}$. The main difference between weakest pre-conditions and Hoare triples is that the former does not mention the required resources in a pre-condition – instead, propositions describing the required resources *implies* the weakest-preconditions. For instance, if one owns the points-to predicate for some location ℓ, then one can derive a weakest-precondition of a load of ℓ. That is, we have that $\ell \hookrightarrow v \vdash \mathsf{wp}\ !\ell\, \{u.u = v * \ell \hookrightarrow v\}$.

The rules for weakest pre-conditions are analogous to the rules for Hoare triples shown in 2.1, so we do not mention them here. Indeed, we may even define the notions in terms of each other. The reason for having both is that it is usually nicer to write specifications in terms of Hoare triples, whereas proofs of weakest pre-conditions are more streamlined.

2.2.3 Later Modality

The presentation we have seen thus far does not allow us to tie propositions to steps in the program; we may want to express that some proposition holds after one program step. To achieve this, Iris provides a *later* modality, written $\triangleright P$. The later modality is technically language independent and hence parallel to program steps, but the way that Hoare triples and weakest pre-conditions are defined ties a single $\triangleright$ to a single step in the program. For instance $\triangleright(\ell \mapsto 5)$ asserts that the location ℓ will contain the value 5 after taking one step in the program. In other words, we do not own the resource that ℓ points to 5 now, but we will own it after one step. In this sense, the later modality weakens propositions; owning a resource now is stronger than owning it later (cf. rule LATER-WEAK).

We update the rules from figure 2.1 to capture the effect that taking a step removes a later. For instance, HT-BETA-LATER is similar to HT-BETA, but it strips away a later in the pre-condition to signify that the program has taken a step (in this case by computing the function application).

The later modality has many uses in Iris, but for our presentation, the Löb induction rule, LÖB, is the most important. It states that, if we want to prove some proposition P, we may assume that we have P later. This rule is really useful when P is a Hoare triple for a recursive function. If f is some recursive function and we want to prove the Hoare triple: $\{Q\}\ f\ u\,\{v.\Phi\ v\}$, then Löb induction gives us an induction hypothesis: $\triangleright(\{Q\}\ f\ u\,\{v.\Phi\ v\})$. As the later modality weakens our proposition, we cannot use it to prove our goal, but after performing the function application using HT-BETA, the $\triangleright$ in our induction hypothesis is stripped away, and we get to assume $\{Q\}\ f\ u\,\{v.\Phi\ v\}$. Hence, if we reach a recursive call inside f, we just have to prove Q, and then we can use our induction hypothesis to prove the recursive call.

LATER-WEAK
$$\frac{Q \vdash P}{Q \vdash \triangleright P}$$

LATER-MONO
$$\frac{Q \vdash P}{\triangleright Q \vdash \triangleright P}$$

LÖB
$$\frac{Q \wedge \triangleright P \vdash P}{Q \vdash P}$$

HT-BETA-LATER
$$\frac{S \vdash \{P\}\, e\, [v/x]\ \{u.Q\}}{S \vdash \{\triangleright P\}\ (\lambda x.e)v\ \{u.Q\}}$$

HT-LOAD
$$\frac{}{S \vdash \{\triangleright \ell \hookrightarrow u\}\ !\ell\,\{v.v = u \wedge \ell \hookrightarrow u\}}$$

HT-STORE
$$\frac{}{S \vdash \{\triangleright \ell \hookrightarrow -\}\ \ell \leftarrow w\,\{v.v = () \wedge \ell \hookrightarrow w\}}$$

2.2.4 Resource Algebra

Thus far, the only kind of resource we have seen are the points-to predicate. This is in itself a very useful notion of resource, but many constructions demand other kinds of resources. Iris allows us to create our own notion of a resource by defining a *resource algebra* which specifies what the resources are, and how the resources interact with each other. Indeed, even the points-to predicate is defined via a resource algebra.

Formally, "A *resource algebra* is a commutative semigroup $\mathcal{M}$ together with a subset $\mathcal{V} \subseteq \mathcal{M}$ of elements called *valid*, and a *partial* function $|\cdot| : \mathcal{M} \to \mathcal{M}$, called the *core*." (Birkedal and Bizjak [2017])

That is, a resource algebra consists of a set of elements (the resources) with an associative and commutative operation, written as $a \cdot b$. The set of elements is usually called the "carrier". Some of the elements are marked as "valid" – only valid resources can be owned in Iris (cf. rule OWN-VALID). Ownership of a non-valid element allows for deriving falsehood.
The core function "extracts" duplicable parts from resources. In particular, if the core of an element a is a itself, i.e. $|a| = a$, then a is freely duplicable. Hence, ownership of a is persistent.
Some resource algebra enjoy the additional property of being *unital*. This essentially means that the carrier contains a unit element ϵ with respect to '$\cdot$' which is both valid and duplicable.
Finally, we note that we can create a pre-order relation for every resource algebra by defining the extension order: $a \preccurlyeq b \iff \exists c, b = a \cdot c$.

Since programs update resources, we will need some way of updating elements in a resource algebra. We write $a \rightsquigarrow b$ to mean that a can be updated to b. Updates of elements should not be able to happen freely; a thread updating its resources should not invalidate the resources that another thread owns. The idea is to ensure that we can only update our resources if we do not make other resources invalid. We call such an update a *frame preserving update*, and define as follows.

FRAME-PRESERVING-UPDATE
$$a \rightsquigarrow b \iff \forall x \in \mathcal{M}, a \cdot x \in \mathcal{V} \implies b \cdot x \in \mathcal{V}.$$

This rule is simplified a little, but it suffices of our purposes. It requires that any other resource that is valid with a is also valid with b. This ensures us that we do not invalidate other elements as a consequence of the update.

Birkedal and Bizjak [2017] shows many examples of basic resource algebra. However, it turns out that often, the desired construction can be realised by *composing* different resource algebra, instead of defining them from the ground up. In fact, in this project, the resource algebras used were exclusively constructed from other resource algebra. Chapter 8 shows the main resource algebras used in this project.
In this report, we isolate the difficulties of creating desired constructions by postulating their existence, and only later, in chapter 8, showing how to realise these using specific resource algebra.

Ghost State

We have until now only discussed resource algebra as an isolated concept. Now we show how resource algebras are tied into the logic of Iris, allowing us to use them in reasoning about programs.

The first component we need is an *update modality*, written $\Rightarrow P$. This modality governs where we are allowed to updated our resources, including the creation of new resources. We present the following three rules for introducing update modalities.

$$\textsc{upd-mono}\ \frac{P \vdash Q}{\Rightarrow P \vdash \Rightarrow Q} \qquad \textsc{upd-intro}\ \frac{}{P \vdash \Rightarrow P} \qquad \textsc{Ht-csq-vs}\ \frac{S \vdash P \Rrightarrow P' \qquad S \vdash \{P'\}\, e\, \{v.\, Q'\} \qquad S \vdash \forall u.\, Q'[u/v] \Rrightarrow Q[u/v]}{S \vdash \{P\}\, e\, \{v.\, Q\}}$$

Here, the *view shift*, $\Rrightarrow$, is defined as $P \Rrightarrow Q = \Box(P \implies \Rightarrow Q)$, and states that, if we own the resources described by P, then we can derive the resources described by Q after updating our resources (for instance via a frame preserving update).

Iris supports reasoning about resource algebra via the notion of *ghost state*: for a resource algebra $\mathcal{M}$ and an element a in the carrier, the assertion $\boxed{a}^{\gamma}$ denotes ownership over an instance of the resource a. Here γ is a *ghost name* – it gives a way to refer specific instances of the resource algebra. The following four rules demonstrate how to create, update, and reason about ghost resources in Iris.

$$\textsc{Ghost-alloc}\ \frac{a \in \mathcal{V}}{\mathsf{True} \vdash \Rightarrow \exists \gamma.\, \boxed{a}^{\gamma}} \qquad \textsc{Ghost-update}\ \frac{a \rightsquigarrow b}{\boxed{a}^{\gamma} \vdash \Rightarrow \boxed{b}^{\gamma}} \qquad \textsc{Own-op}\ \frac{}{\boxed{a}^{\gamma} * \boxed{b}^{\gamma} \dashv\vdash \boxed{a \cdot b}^{\gamma}} \qquad \textsc{Own-valid}\ \frac{}{\boxed{a}^{\gamma} \vdash a \in \mathcal{V}}$$

2.2.5 Invariants

The last feature of Iris we need is that of *invariants*. Some resources are required by multiple threads, but those resources may not be persistent, and hence not duplicable. To get around this, one can devise an invariant for said resources – a proposition P, which describes the resources in a way that is always true. Then, one can assert that this proposition is an invariant, written $\boxed{P}^{\mathcal{N}}$, and since invariant are persistent, they can be given to multiple threads. The threads may then access the resources inside the invariant by *opening* it. There are three criteria when opening an invariant. Firstly, the invariant can only be open for one program step. This is enforced by making the opening rule for invariants require that the expression in the Hoare triple or weakest pre-condition is "atomic". We can usually always satisfy this criteria by applying the Ht-bind rule.
The second criteria is that the invariant can only be opened once, before being closed again. Iris enforces this by attaching masks to many of the constructs presented in the previous sections. The details of masks are not important for our presentation. Masks tell us which invariants we are allowed to open. For instance, the invariant opening rule for weakest pre-conditions, wp-inv-open-namespace, attaches the mask $\mathcal{E} \setminus \mathcal{N}^{\uparrow}$ to the weakest pre-condition, signifying that we cannot use invariants in the namespace $\mathcal{N}$ to prove the weakest precondition.
Finally, as wp-inv-open-namespace also shows, we must prove that the invariant still holds after e has taken its one step (this is realised by having P in the post-condition). The reason is that other threads might also rely on the invariant, hence we have to reinstate it immediately.

INV-ALLOC
$$\triangleright P \vdash \Rrightarrow_{\emptyset} \boxed{P}^{\mathcal{N}}$$

WP-INV-OPEN-NAMESPACE
$$\frac{e \text{ is an atomic expression} \qquad \mathcal{N}^{\uparrow} \subseteq \mathcal{E}}{\boxed{P}^{\mathcal{N}} * \left(\triangleright P \twoheadrightarrow \mathsf{wp}_{\mathcal{E} \setminus \mathcal{N}^{\uparrow}}\, e\, \{v. \triangleright P * \Phi(v)\}\right) \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}}$$

A technicality of invariants is that we only get the resources *later* when opening them.[1] This is usually not an issue as most of the rules for stepping through programs only require resources later (cf. section 2.2.3). Hence we will usually assume that we have resources available *now*, when we open invariants. We mention the later explicitly in the few cases where only getting the resources later is important.

2.2.6 Locks

The Two-Lock M&S Queue uses locks, so we discuss these briefly in this section. A lock has three functions: newLock, acquire, and release. In Iris, locks protect resources; when we create a new lock we give it the resources it must protect. When a thread acquires the lock, the thread gets access to the protected resources. Releasing the locks then requires that the resources are transferred back to the lock. In our project, we use the following specification for locks from Birkedal and Bizjak [2017] (example 8.38 at time of writing).

$$\begin{aligned}
&\exists\, \text{isLock} : \mathit{Val} \to \mathsf{Prop} \to \mathsf{GhostName} \to \mathsf{Prop}.\\
&\exists\, \text{locked} : \mathsf{GhostName} \to \mathsf{Prop}.\\
&\quad \square(\forall P, v, \gamma.\ \text{isLock}(v, P, \gamma) \implies \square\, \text{isLock}(v, P, \gamma))\\
\wedge\ &\square(\forall \gamma.\ \text{locked}(\gamma) * \text{locked}(\gamma) \implies \mathsf{False})\\
\wedge\ &\forall P.\ \{P\}\ \text{newLock}()\ \{v. \exists \gamma.\ \text{isLock}(v, P, \gamma)\}\\
\wedge\ &\forall P, v, \gamma.\ \{\text{isLock}(v, P, \gamma)\}\ \text{acquire}\, v\ \{_. P * \text{locked}(\gamma)\}\\
\wedge\ &\forall P, v, \gamma.\ \{\text{isLock}(v, P, \gamma) * P * \text{locked}(\gamma)\}\ \text{release}\, v\ \{_. \mathsf{True}\}
\end{aligned}$$

The specification assert the existence of two predicates: the lock predicate, "isLock" and a lock token, "locked". The lock predicate describes that a value represents a lock, and governs which resources the lock protects. The specification for the newLock function states that if we own the resources described by P, we can create a new lock by invoking newLock. The value returned by the function will then represent the lock, and it protects the resources described by P. The lock predicate is persistent so we can give the lock predicate to multiple threads, allowing them to use the specifications for acquire and release. The specification for acquire grants us access to the resources that the lock is protecting as well as a non-duplicable token, locked(γ). This token tells us that we are the sole owner of the lock. To release the lock, we must give up the locked(γ) token and the resources the lock protects.

2.3 Formalisation in Coq

Iris has been mechanised in the Coq proof assistant[2] – a tool to machine-check proofs of mathematical assertions. All results in this project have been completely machine-verified in the Iris mechanisation in Coq. Specifications in the Coq formalisation of Iris are usually written in terms of Hoare triples but proved by first converting the Hoare triples to equivalent weakest-preconditions, as this is usually easier to work with. The proofs presented in this report will follow suit and give specifications using Hoare triples, but prove them assuming they are weakest-preconditions. The proofs presented in the report thus follow the mechanised proofs very closely, making it possible to "step-through" the mechanised proofs in tandem with reading the paper-proofs presented in this report.

One caveat is that there is somewhat of a discrepancy between Iris on paper, and the Iris formalisation in Coq. Working with the latter requires a bit deeper understanding of the model of Iris. Jung et al. [2018] explains the underlying model of an older version of Iris, but many of the concepts discussed are still relevant.[3]

[1] Getting the resources immediately would be unsound.
[2] The mechanisation can be found at `https://gitlab.mpi-sws.org/iris/iris/`
[3] For an up-to-date presentation, consult the Technical Reference at `https://iris-project.org/`

Table 2.1 gives an overview of the files developed in the project and how they relate to this report. All files related to the project are available at `https://github.com/MatteP1/thesis`.

File Name	Relevant Sections	Description
`queue_specs.v`	Chapter 3	Implementation-independent specifications and derivations.
`twoLockMSQ_derived.v`	Section 3.5	
`lockFreeMSQ_derived.v`	Section 3.5	
`MSQ_common.v`	Chapters 4 - 7	Common definitions and lemmas.
`twoLockMSQ_impl.v`	Chapter 4	Two-Lock M&S Queue implementation and proofs of sequential, concurrent, and Hocap-style specifications.
`twoLockMSQ_sequential_spec.v`	Section 5.2	
`twoLockMSQ_concurrent_spec.v`	Section 5.3	
`twoLockMSQ_hocap_spec.v`	Section 5.4	
`lockFreeMSQ_impl.v`	Chapter 6	Lock-Free M&S Queue implementation and Hocap-style specification proof.
`lockFreeMSQ_hocap_spec.v`	Chapter 7	
`lockAndCCFreeMSQ_impl.v`	Section 7.6	Consistency-Check-Free version of Lock-Free M&S Queue.
`lockAndCCFreeMSQ_hocap_spec.v`	Section 7.6	

Table 2.1: Overview of Coq Files.

2.3.1 Compiling the Project

Compiling the project requires both Coq and Iris to be installed. Once installed, open a terminal and navigate to the project folder, `/thesis`, which contains the `_CoqProject` file. Here, run `make`. This will both create a Coq Makefile and run it. The project is known to compile with Coq version 8.19.0 and Iris version 4.2.0.

Chapter 3

Queue Specifications

3.1 Specifications for Queues

In this chapter we discuss some possible specifications for queue data structures in general. As such, we need to make some basic assumption about what we expect from a queue. Firstly, we adopt the convention that a queue consists of three functions: initialize, enqueue, and dequeue. Exactly what these functions do will depend on the specific implementation of the queue, but we give some general pointers to what we expect of them.
The initialize function should create a queue which is initially empty. The functions enqueue and dequeue can then be invoked subsequently on said queue.
In addition to being parametrised on the queue, the enqueue function should also take a value as input. When invoking enqueue with such a value, the function should add this value to the end of the queue.
The dequeue function will attempt to dequeue an element from the queue. Since queues are allowed to be empty, the dequeue function is assumed to return an option value. If the queue is empty, then dequeue will return None, and otherwise it will remove an element from the front of the queue, and return this wrapped in a Some.

Working in a concurrent setting often gets quite complicated quite fast, and proving that ones queue satisfies the above desirables can become quite tricky. In fact, even defining those qualities formally can become non-trivial. As such, we give three different specifications for queues. In section 3.2 we give a specification that assumes the queue is run in a sequential setting. Next, in section 3.3, we give a specification that *does* allow for concurrency but gives up on some of the above qualities. Primarily, it does not track the contents of the queue; invoking enqueue on a value doesn't guarantee that the value is added to the queue. Finally, in section 3.4 we give a specification that allows for both concurrency *and* tracking of the contents of the queue.

3.2 Defining a Sequential Specification

Let us first consider a specification in the simple case where we do not 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. The specification we give will track 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. That is, $xs_v : \mathit{List\ Val}$.
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.

To allow queues to use whichever ghost names they like, we introduce the type "$\mathit{SeqQgnames}$" whose purpose is to keep track of the ghost names used for a specific queue. One may think of $\mathit{SeqQgnames}$ as set of fixed-length tuples of ghost names.

With this, we give our definition of a sequential specification for a queue.

Definition 3.2.1 (Sequential Specification)**.**

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

The proposition $\mathrm{isQueue_S}(v_q, xs_v, G)$ captures that the value v_q is a queue, whose contents matches that of our abstract representation xs_v, and the queue uses the ghost names described by G. Clients of a queue satisfying this specification can then apply the three Hoare triples at invocations for initialize, enqueue, and dequeue, which tells them how the queue changes as a result of the invocation.
Note that the $\mathrm{isQueue_S}$ predicate is not required to be persistent, hence it cannot be duplicated and given to multiple threads. Since this predicate is required to apply the Hoare triples, then only a single thread can use the specification at a time. This is the sense in which the specification is sequential.

3.3 Defining a Concurrent Specification

As discussed in the previous section, a concurrent specification will need the queue predicate to be duplicable to allow multiple threads to use the specification concurrently. For this specification, we denote the queue predicate by $\mathrm{isQueue_C}$.

To achieve duplicability of $\mathrm{isQueue_C}$, we shall give up on tracking the abstract state of the queue. The reason for doing so is the following. Consider the case where we own the queue predicate and it states that the abstract state of queue is xs_v. We spawn two threads and give them both the queue predicate stating that the abstract value is xs_v. Now they both invoke enqueue, one with value v and the other with v'. This make one thread conclude that the abstract state of the queue is $v \mathbin{+\!\!+} xs_v$, and the other that $v' \mathbin{+\!\!+} xs_v$, whereas in reality, the queue contains both v and v'. This is of course not desirable, and it is not immediately obvious how to solve this issue.

We *can* however make the queue guarantee that a given property holds of all the values in the queue. We do this by parametrising $\mathrm{isQueue_C}$ with a predicate, Ψ, which it will maintain holds for all elements in the queue. In this way, when dequeueing, we at least know that if we get some value, then Ψ holds of this value. The specification we wish to prove is as follows.

Definition 3.3.1 (Concurrent Specification)**.**

$$
\begin{aligned}
&\exists\, \mathrm{isQueue_C} : (\mathit{Val} \to \mathsf{Prop}) \to \mathit{Val} \to \mathit{ConcQgnames} \to \mathsf{Prop}.\\
&\forall \Psi : \mathit{Val} \to \mathsf{Prop}.\\
&\quad \forall v_q, G.\ \mathrm{isQueue_C}(\Psi, v_q, G) \implies \Box\, \mathrm{isQueue_C}(\Psi, v_q, G)\\
\wedge\ &\quad \{\mathsf{True}\}\ \mathrm{initialize}\ ()\ \{v_q.\exists G.\ \mathrm{isQueue_C}(\Psi, v_q, G)\}\\
\wedge\ &\quad \forall v_q, v, G.\ \{\mathrm{isQueue_C}(\Psi, v_q, G) * \Psi(v)\}\ \mathrm{enqueue}\ v_q\ v\ \{w.\mathsf{True}\}\\
\wedge\ &\quad \forall v_q, G.\ \{\mathrm{isQueue_C}(\Psi, v_q, G)\}\ \mathrm{dequeue}\ v_q\ \{w.w = \mathrm{None} \vee (\exists v.\ w = \mathrm{Some}\, v * \Psi(v))\}
\end{aligned}
$$

This specification additionally requires that $\mathrm{isQueue_C}$ is persistent, which means that it can be duplicated and given to multiple threads. We again use a collection of ghost names, which we here denote $\mathit{ConcQgnames}$.

3.4 Defining a HOCAP-style Specification

In this section we will explore our most general specification: one that allows for both concurrency and tracking of the contents of the queue. We refer to this specification as a Hocap-style specification – Higher Order Concurrent Abstract Predicate – since it will be concurrent and parametrised by abstract

predicates. This specification is more general than both the sequential and concurrent specifications in the sense that they are derivable from the Hocap-style specification. We prove this in section 3.5.

As with concurrent specification, we cannot simply parametrise the queue predicate (now denoted isQueue) with the abstract state of the queue. So to allow clients to keep track of the contents of the queue, we will "split" the abstract state into two parts; the authoritative view and the fragmental view. The clients will then own the fragmental view, allowing them to keep track of the contents of the queue, whereas the isQueue 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 if one possess both the authoritative and fragmental views. Hence, clients will have to supply the fragmental view to be able to apply the specifications for enqueue and dequeue.

For an abstract state xs_v and a ghost name γ, we shall use the notation $\gamma \Mapsto_\bullet xs_v$ to mean that the authoritative view of the abstract state associated with γ is xs_v. Similarly we write $\gamma \Mapsto_\circ xs_v$ to mean that the fragmental view associated with γ is xs_v.

We introduce three lemmas to help working with these predicates. These are proved in section 8.2. The first lemma shows that we can create fresh authoritative and fragmental views for any abstract state. The abstract state will usually be empty when allocating.

Lemma 1 (Abstract State Alloc)**.** For any abstract state xs_v, we have

$$\vdash \Rrightarrow \exists \gamma.\, \gamma \Mapsto_\bullet xs_v * \gamma \Mapsto_\circ xs_v$$

The second shows that the authoritative and fragmental views of the abstract state agree.

Lemma 2 (Abstract State Agree)**.** For a ghost name γ and abstract states xs_v and xs'_v, we have

$$\gamma \Mapsto_\bullet xs'_v * \gamma \Mapsto_\circ xs_v \vdash xs_v = xs'_v$$

The final lemma shows that, if we own both the authoritative and fragmental views, we are allowed to update the abstract state to whatever we like.

Lemma 3 (Abstract State Update)**.** For any ghost name γ, and abstract values xs_v, xs'_v, and xs''_v, we have

$$\gamma \Mapsto_\bullet xs'_v * \gamma \Mapsto_\circ xs_v \Rrightarrow \gamma \Mapsto_\bullet xs''_v * \gamma \Mapsto_\circ xs''_v$$

The collection of ghost names is now denoted $Qgnames$, but as all queues will have to deal with the ghost name keeping track of the abstract state as describe above, we require that any $Qgnames$ will contain a ghost name used for this purpose. We shall refer to this ghost name as γ_{Abst}.

With this, we now give the Hocap-style specification, and explain the intricacies of it afterwards.

Definition 3.4.1 (Hocap Specification)**.**

$$\begin{aligned}
&\exists\, \text{isQueue} : \mathit{Val} \to \mathit{Qgnames} \to \mathsf{Prop}.\\
&\quad \forall v_q, G.\ \text{isQueue}(v_q, G) \implies \square\, \text{isQueue}(v_q, G)\\
\wedge\ &\quad \{\mathsf{True}\}\ \text{initialize}\ ()\ \{v_q. \exists G.\ \text{isQueue}(v_q, G) * G.\gamma_{\text{Abst}} \Mapsto_\circ []\}\\
\wedge\ &\quad \forall v_q, v, G, P, Q.\ \left(\forall xs_v.\, G.\gamma_{\text{Abst}} \Mapsto_\bullet xs_v * P \Rrightarrow_{\mathcal{E}\setminus\mathcal{N}.i\uparrow} \triangleright\, G.\gamma_{\text{Abst}} \Mapsto_\bullet (v :: xs_v) * Q\right) \mathrel{-\!\!*}\\
&\qquad\qquad \{\text{isQueue}(v_q, G) * P\}\ \text{enqueue}\ \ v_q\ v\ \{w.Q\}\\
\wedge\ &\quad \forall v_q, G, P, Q.\\
&\qquad \left(\forall xs_v.\, G.\gamma_{\text{Abst}} \Mapsto_\bullet xs_v * P \Rrightarrow_{\mathcal{E}\setminus\mathcal{N}.i\uparrow} \triangleright \left(\begin{array}{l}(xs_v = [] * G.\gamma_{\text{Abst}} \Mapsto_\bullet xs_v * Q(\text{None}))\\ \vee \left(\begin{array}{l}\exists v, xs'_v.\, xs_v = xs'_v \mathbin{+\!\!+} [v] *\\ G.\gamma_{\text{Abst}} \Mapsto_\bullet xs'_v * Q(\text{Some}\, v)\end{array}\right)\end{array}\right)\right) \mathrel{-\!\!*}\\
&\qquad \{\text{isQueue}(v_q, G) * P\}\ \text{dequeue}\ \ v_q\ \{w.Q(w)\}
\end{aligned}$$

As for the concurrent specification, we require that the queue predicate, isQueue, is persistent, giving us support for concurrent clients.

Next, the specification for initialize gives clients an additional resource in the postcondition: the ownership of the fragmental view of the empty list, $G.\gamma_{\text{Abst}} \Mapsto_\circ []$. As discussed above, this allows them to keep track of the contents of the queue.

The specifications for enqueue and dequeue have now 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 associated Hoare triples. However, before the client gets access to the Hoare triple for enqueue or dequeue they must prove a view-shift. This view-shift 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 of the view-shift contains a $\triangleright$. This signifies that the update in the abstract state is tied to a step in the code. The mask on the view-shift further disallows opening of invariants in the namespace $\mathcal{N}.i$. This is to allow queues that use invariants to apply the view-shifts while their invariants are open (their invariants must of course be within the namespace $\mathcal{N}.i$).

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 view-shift, clients are not 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 view-shift) in conjunction with updating the concrete view.

Exactly how client supply the fragmental view depends on what the client wants to achieve. We will see two options, when we derive the sequential and concurrent specifications from this Hocap-style specification in the next section.

3.5 Deriving Sequential and Concurrent Specifications

It is technically possible to derive the sequential and concurrent specifications from the Hocap-style specification without having proven the Hocap-style specification for a specific queue. However, it might be beneficial for the reader to first see how we can prove each specification *directly* for a specific queue, which we show in chapter 5, and then return to this section.

In this section we show that we can derive the sequential and concurrent specifications from sections 3.2 and 3.3 from the Hocap-style specification we saw in the previous chapter. These derivations are implementation independent, so we assume that we have some functions initialize, enqueue, and dequeue which satisfy the Hocap-style specification of definition 3.4.1, and we wish to prove that they also satisfy the sequential and concurrent specifications of definitions 3.2.1 and 3.3.1. That is, we assume that we have a collection of ghost names, *Qgnames*, a persistent queue predicate, isQueue, and the three Hocap-style specifications for initialize, enqueue, and dequeue. Both derivations will simply use *Qgnames* as the collection of ghost names. So we let *SeqQgnames* and *ConcQgnames* be *Qgnames* in the following.

3.5.1 Deriving the Sequential Specification

Recall the sequential specification specified in definition 3.2.1. It demands a queue predicate, isQueue_S. We here choose to define it as follows.

Definition 3.5.1 (isQueue_S Predicate (Derive))**.**

$$\begin{aligned}\text{isQueue}_\text{S}(v_q, xs_v, G) \triangleq\ & \text{isQueue}(v_q, G) * \\ & G.\gamma_{\text{Abst}} \Rightarrow_\circ xs_v\end{aligned}$$

We proceed to prove the sequential specifications for the three queue functions.

Sequential Initialise Specification

Recall the sequential specification for initialise:

$$\{\mathsf{True}\}\ \text{initialize}\ ()\ \{v_q.\exists G.\ \text{isQueue}_\text{S}(v_q, [], G)\}$$

Unfolding isQueue_S, this Hoare triple follows directly from the Hocap-style initialise specification (cf. lemma 3.4.1).

Sequential Enqueue Specification

The specification for enqueue states:

$$\forall v_q, v, xs_v, G.\, \{\text{isQueue}_\text{S}(v_q, xs_v, G)\}\ \text{enqueue}\ v_q\ v\, \{w.\, \text{isQueue}_\text{S}(v_q, (v :: xs_v), G)\}$$

So assume some v_q, v, xs_v, and G. Unfolding isQueue_S, our goal becomes:

$$\{\text{isQueue}(v_q, G) * G.\gamma_\text{Abst} \Mapsto_\circ xs_v\}\ \text{enqueue}\ v_q\ v\, \{w.\, \text{isQueue}(v_q, G) * G.\gamma_\text{Abst} \Mapsto_\circ (v :: xs_v)\}$$

To prove the Hoare triple, we shall use the Hocap-style specification for enqueue. This however requires us to pick P and Q, and prove the resulting view-shift. We choose

$$P \triangleq G.\gamma_\text{Abst} \Mapsto_\circ xs_v \qquad\qquad Q \triangleq G.\gamma_\text{Abst} \Mapsto_\circ (v :: xs_v)$$

and assume some xs'_v. We must then prove the view-shift:

$$G.\gamma_\text{Abst} \Mapsto_\bullet xs'_v * G.\gamma_\text{Abst} \Mapsto_\circ xs_v \Rrightarrow_{\mathcal{E}\setminus\mathcal{N}.i\uparrow} \triangleright\, G.\gamma_\text{Abst} \Mapsto_\bullet (v :: xs'_v) * G.\gamma_\text{Abst} \Mapsto_\circ (v :: xs_v)$$

Assume $G.\gamma_\text{Abst} \Mapsto_\bullet xs'_v$ and $G.\gamma_\text{Abst} \Mapsto_\circ xs_v$. By lemma 2, $xs_v = xs'_v$, hence, we can apply lemma 3 to update the authoritative and fragmental views to $(v :: xs_v)$, which exactly proves the consequent of the view-shift.[1]

With this, we now get access to the following Hoare triple:

$$\{\text{isQueue}(v_q, G) * G.\gamma_\text{Abst} \Mapsto_\circ xs_v\}\ \text{enqueue}\ v_q\ v\, \{w. G.\gamma_\text{Abst} \Mapsto_\circ (v :: xs_v)\} \tag{3.1}$$

The pre-condition already matches our goal, so we just have to get the post-condition to match. To do this, we must get $\text{isQueue}(v_q, G)$ in the post-condition of 3.1, but since is $\text{isQueue}(v_q, G)$ persistent and we have it in the pre-condition, we may also assume it in post-condition.

Sequential Dequeue Specification

We use a similar approach to above to prove the sequential dequeue specification:

$$\begin{array}{l}\forall v_q, xs_v, G.\, \{\text{isQueue}_\text{S}(v_q, xs_v, G)\} \\ \qquad \text{dequeue}\ v_q \\ \qquad \left\{w.\ \begin{array}{l}(xs_v = [] * w = \text{None} * \text{isQueue}_\text{S}(v_q, xs_v, G)) \lor \\ (\exists v, xs'_v.\, xs_v = xs'_v \mathbin{+\!\!+} [v] * w = \text{Some}\, v * \text{isQueue}_\text{S}(v_q, xs'_v, G))\end{array}\right\}\end{array}$$

So we assume some v_q, xs_v, G. We now instantiate the Hocap-style dequeue specification with the following choices:

$$\begin{array}{rl} P \triangleq & G.\gamma_\text{Abst} \Mapsto_\circ xs_v \\ Q(w) \triangleq & \begin{array}{l}(xs_v = [] * w = \text{None} * G.\gamma_\text{Abst} \Mapsto_\circ xs_v) \lor \\ (\exists v, xs'_v.\, xs_v = xs'_v \mathbin{+\!\!+} [v] * w = \text{Some}\, v * G.\gamma_\text{Abst} \Mapsto_\circ xs'_v)\end{array}\end{array}$$

We must now prove the resulting view-shift to get the Hoare triple (note that we haven't substituted in P and Q for the sake of readability). So assume some xs'_v. We must show:

$$G.\gamma_\text{Abst} \Mapsto_\bullet xs'_v * P \Rrightarrow_{\mathcal{E}\setminus\mathcal{N}.i\uparrow} \triangleright \left(\begin{array}{l}(xs'_v = [] * G.\gamma_\text{Abst} \Mapsto_\bullet xs'_v * Q(\text{None})) \\ \lor \left(\begin{array}{l}\exists v, xs''_v.\, xs'_v = xs''_v \mathbin{+\!\!+} [v] * \\ G.\gamma_\text{Abst} \Mapsto_\bullet xs''_v * Q(\text{Some}\, v)\end{array}\right)\end{array}\right)$$

By lemma 2, xs'_v must be equal to xs_v. We do a case analysis on xs_v. If xs_v is empty, we prove the left disjunct in the consequent of the view-shift, *without* updating the authoritative and fragmental views. If xs_v is non-empty, i.e. $xs_v = xs''_v \mathbin{+\!\!+} [v]$ for some xs''_v and v, then we prove the right-side of the consequent in the view-shift by using lemma 3 to update the authoritative and fragmental views to the new abstract state, xs''_v.

[1] The consequent technically has a $\triangleright$, but proving something *now* is stronger than proving it *later*.

With this, we get access to the Hoare triple (now with P and Q substituted in):

$$\begin{array}{l}\{\text{isQueue}(v_q, G) * G.\gamma_{\text{Abst}} \Mapsto_\circ xs_v\} \\ \quad \text{dequeue } v_q \\ \left\{w.\ \begin{array}{l}(xs_v = [] * w = \text{None} * G.\gamma_{\text{Abst}} \Mapsto_\circ xs_v) \vee \\ (\exists v, xs'_v.\ xs_v = xs'_v \mathbin{+\!\!+} [v] * w = \text{Some}\, v * G.\gamma_{\text{Abst}} \Mapsto_\circ xs'_v)\end{array}\right\}\end{array} \tag{3.2}$$

As before, the only difference between this Hoare triple and the one we must prove is that we are missing $\text{isQueue}(v_q, G)$ in the post-condition. We can again get this from the fact that the queue predicate is persistent.

3.5.2 Deriving the Concurrent Specification

We prove the concurrent specification of definition 3.3.1. Remember that we need the isQueue_C predicate to be persistent, hence we cannot simply assert $G.\gamma_{\text{Abst}} \Mapsto_\circ xs_v$ as we did for isQueue_S.[2] Instead, we will put it into an invariant. The queue predicate we will use looks as follows.

Definition 3.5.2 (isQueue_C Predicate (Derive))**.**

$$\text{isQueue}_\text{C}(\Psi, v_q, G) \triangleq \text{isQueue}(v_q, G) * \boxed{\exists xs_v.\ G.\gamma_{\text{Abst}} \Mapsto_\circ xs_v * \text{All}(xs_v, \Psi)}^{\mathcal{N}.c}$$

Persistency of isQueue_C follows by the persistency of isQueue and the fact that invariants are persistent.

Concurrent Initialise Specification

We have to derive the specification:

$$\{\mathsf{True}\}\ \text{initialize}\ ()\ \{v_q.\exists G.\ \text{isQueue}_\text{C}(\Psi, v_q, G)\}$$

As before, this specification only differs from the Hocap-style specification for initialise in the post-condition. We use the *generalised* rule of consequence HT-CSQ-VS and show that the post-condition of the Hocap-style specification, i.e. $\exists G.\ \text{isQueue}(v_q, G) * G.\gamma_{\text{Abst}} \Mapsto_\circ []$, implies the post-condition above but with an update modality, $\Rrightarrow$, in front. Both mention the queue predicates, $\text{isQueue}(v_q, G)$, so it suffices to prove the invariant:

$$\Rrightarrow \boxed{\exists xs_v.\ G.\gamma_{\text{Abst}} \Mapsto_\circ xs_v * \text{All}(xs_v, \Psi)}^{\mathcal{N}.c}$$

We have $G.\gamma_{\text{Abst}} \Mapsto_\circ []$ from the Hocap-style post-condition, and $\text{All}([], \Psi)$ is equivalent to True. Hence, we can deduce $\exists xs_v.\ G.\gamma_{\text{Abst}} \Mapsto_\circ xs_v * \text{All}(xs_v, \Psi)$. The rule INV-ALLOC shows us that we can turn this into the above invariant, so we are done.

Concurrent Enqueue Specification

We must derive:

$$\forall v_q, v, G.\ \{\text{isQueue}_\text{C}(\Psi, v_q, G) * \Psi(v)\}\ \text{enqueue}\ v_q\ v\ \{w.\mathsf{True}\}$$

Assume some v_q, v, G, and the invariant $\boxed{\exists xs_v.\ G.\gamma_{\text{Abst}} \Mapsto_\circ xs_v}^{\mathcal{N}.c}$. Our goal becomes the following Hoare triple:

$$\{\text{isQueue}(v_q, G) * \Psi(v)\}\ \text{enqueue}\ v_q\ v\ \{w.\mathsf{True}\}$$

We specialise the Hocap-style enqueue specification with $P \triangleq \Psi(v)$ and $Q \triangleq \mathsf{True}$. With this choice, the Hoare triple we get after proving the view-shift exactly matches our goal. Hence, we are done if we can prove the view-shift.

We assume xs'_v, and prove the view-shift:

$$G.\gamma_{\text{Abst}} \Mapsto_\bullet xs'_v * \Psi(v) \Rrightarrow_{\mathcal{E}\setminus\mathcal{N}.i\uparrow} \triangleright\, G.\gamma_{\text{Abst}} \Mapsto_\bullet (v :: xs'_v) * \mathsf{True}$$

[2] As we explain in section 8.2, no elements of the abstract state resource algebra are persistent.

So assume $G.\gamma_{\text{Abst}} \Mapsto_\bullet xs'_v$ and $\Psi(v)$. We open the invariant giving us $G.\gamma_{\text{Abst}} \Mapsto_\circ xs_v$ and $\text{All}(xs_v, \Psi)$ for some xs_v. By lemma 2 we know that $xs_v = xs'_v$. We now update the abstract state to $(v :: xs_v)$ using lemma 3, obtaining $G.\gamma_{\text{Abst}} \Mapsto_\bullet (v :: xs_v)$ and $G.\gamma_{\text{Abst}} \Mapsto_\circ (v :: xs_v)$. We use the first to prove the consequent of the view-shift. Before we are done, we must close the invariant. We use the fragmental part together with $\text{All}(xs_v, \Psi)$ and $\Psi(v)$ to do this.

Concurrent Dequeue Specification

Finally, we derive the concurrent specification for dequeue.

$$\forall v_q, G.\, \{\text{isQueue}_{\text{C}}(\Psi, v_q, G)\}\ \text{dequeue}\ \ v_q\, \{w.w = \text{None} \lor (\exists v.\, w = \text{Some}\, v * \Psi(v))\}$$

So we assume some v_q, G, G, and the invariant. Our goal is now:

$$\{\text{isQueue}(v_q, G)\}\ \text{dequeue}\ \ v_q\, \{w.w = \text{None} \lor (\exists v.\, w = \text{Some}\, v * \Psi(v))\}$$

We make the following choices for P and Q when instantiating the specification for dequeue:

$$P \triangleq \mathsf{True} \qquad\qquad Q(w) \triangleq w = \text{None} \lor (\exists v.\, w = \text{Some}\, v * \Psi(v))$$

Again, with this choice, the Hoare triple we get after proving the view-shift exactly matches our goal.

We assume some xs'_v, and prove the view-shift (again, without substituting in P and Q):

$$G.\gamma_{\text{Abst}} \Mapsto_\bullet xs'_v * P \Rrightarrow_{\mathcal{E}\setminus\mathcal{N}.i\uparrow} \triangleright \left(\begin{array}{l} (xs'_v = [] * G.\gamma_{\text{Abst}} \Mapsto_\bullet xs'_v * Q(\text{None})) \\ \lor \left(\begin{array}{l} \exists v, xs''_v.\, xs'_v = xs''_v \mathbin{+\!\!+} [v] * \\ G.\gamma_{\text{Abst}} \Mapsto_\bullet xs''_v * Q(\text{Some}\, v) \end{array} \right) \end{array} \right)$$

Assume $G.\gamma_{\text{Abst}} \Mapsto_\bullet xs'_v$. Opening the invariant, we get $\text{All}(xs_v, \Psi)$ and the fragmental part, $G.\gamma_{\text{Abst}} \Mapsto_\circ xs_v$, for some xs_v, which by lemma 2 we know is equal to xs'_v. We proceed by case analysis on xs_v. If it is empty, we simply close the invariant again, and proceed to prove the first disjunct of the consequent. If it is not empty, then we have $xs_v = xs''_v \mathbin{+\!\!+} [v]$ for some xs''_v and v. We use lemma 3 to update the abstract state to xs''_v, and split $\text{All}(xs_v, \Psi)$ into $\Psi(v)$ and $\text{All}(xs''_v, \Psi)$. Using this and the fragmental part, we close the invariant again. To finish, we prove the right disjunct of the consequent using the authoritative part and $\Psi(v)$.[3]

[3]Note here the importance of the $\triangleright$ in the consequent. From the invariant, we technically only have $\triangleright\, \Psi(v)$. If we didn't have the later in the consequent, we would have to prove $\Psi(v)$ from $\triangleright\, \Psi(v)$ which isn't possible.

Chapter 4

The Two-Lock Michael-Scott Queue

In this chapter, we give an implementation of the blocking version of the M&S Queue, the Two-Lock M&S Queue, in HeapLang. This implementation differs slightly from the original, presented in Michael and Scott [1996], but most changes simply reflect the differences in the two languages.

4.1 Introduction

This queue uses two locks to allow for enqueues and dequeues to happen concurrently; one lock protects the enqueue function, and another lock protects the dequeue function. The idea is to exploit the fact that removing elements through dequeue and adding elements through enqueue are largely orthogonal operations, that do not clash with each other. Further, dequeues happen in one end of the queue and enqueues in the other, so often they will operate on separate resources. When the queue is empty and they operate on the same resources, it becomes less trivial as to why this approach is safe. We shall explore this case in detail later.

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 *head* node, marking the beginning of the queue. Note that the head 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 head node, and a tail pointer (ℓ_{tail}) which points to some node in the linked list, denoted the tail node.

In my implementation, a node is a triple $(\ell_{i\_\text{in}}, w_i, \ell_{i\_\text{out}})$ satisfying that location ℓ_{i_in} points to the pair $(w_i, \ell_{i\_\text{out}})$. Here, w_i either contains the value of the node v_i wrapped in a Some (i.e. $w_i = \text{Some}\, v_i$) or it is None. 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 $(\ell_{i\_\text{in}}, w_i, \ell_{i\_\text{out}})$, we mean that $\ell \mapsto \ell_{i\_\text{in}}$. Hence, if we have two adjacent nodes $(\ell_{i\_\text{in}}, w_i, \ell_{i\_\text{out}})$, $(\ell_{i+1\_\text{in}}, w_{i+1}, \ell_{i+1\_\text{out}})$ in the linked list, then we have the following structure: $\ell_{i\_\text{in}} \mapsto (w_i, \ell_{i\_\text{out}})$, $\ell_{i\_\text{out}} \mapsto \ell_{i+1\_\text{in}}$, and $\ell_{i+1\_\text{in}} \mapsto w_{i+1}, \ell_{i+1\_\text{out}}$. For a given triple $x = (\ell_{\text{in}}, w, \ell_{\text{out}})$, we introduce the following notation:

$$\text{in}(x) = \ell_{\text{in}} \qquad \text{val}(x) = w \qquad \text{out}(x) = \ell_{\text{out}}$$

This way of defining nodes essentially means that the "in" pointer becomes a sort of *identifier* for the node. That is, if we have two nodes x, x', and they agree on the "in" pointer, $\text{in}(x) = \text{in}(x')$, then they are in fact the same node, $x = x'$. We capture this property formally in lemma 33, which can be found in the appendix.

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. This comes down to difference between HeapLang and the C-like language used in the original implementation [Michael and Scott, 1996]. Variables in the C-like language are technically just locations, and the assignment operator for variables simply corresponds to a store operation. In HeapLang, variables are modelled directly as locations which gives us an apparent extra pointer indirection compared to the original implementation.

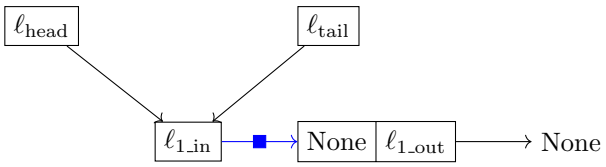


Figure 4.1: Queue after initialisation.

4.2 Implementation

The queue consists of 3 functions: initialize, enqueue, and dequeue. Their implementation is shown in figure 4.4. 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 which we discussed in section 2.2.6.

4.2.1 Initialise

initialize will first create a single node – the head node – marking the start of the linked list. It then creates two locks: the head lock, denoted h_{lock}, protecting the head pointer, and the tail lock, denoted t_{lock}, protecting the tail pointer. Finally, it creates the head and tail pointers, both pointing to the head node. The queue is then a pointer to a structure containing the head and tail pointers, and the two locks.

Figure 4.1 illustrates the structure of the queue after initialisation. Note that one of the pointers is decorated with a square. 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 $(w_i, \ell_{i\_\text{out}})$. We shall use the notation $\ell \mapsto^{\square} v$ (introduced in Vindum and Birkedal [2021]) to mean that ℓ points to v persistently.

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

4.2.2 Enqueue

To enqueue a value, we 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 4.2.

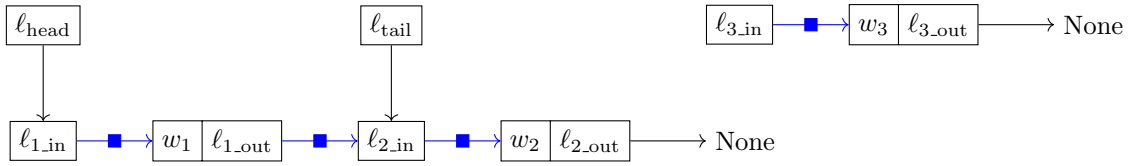
enqueue takes as argument the value to be enqueued and creates a new node containing this value (corresponding to figure 4.2a). This creation doesn't interact with the underlying queue data-structure, hence why we don't acquire t_{lock} first. After creating the new node, we must make the last node in the linked list point to it. Since this operation interacts with the queue, we first acquire t_{lock}. Once we obtain the lock, we make the last node in the linked list point to our new node (figure 4.2b). Following this, we swing ℓ_{tail} to the newly inserted node (figure 4.2c).

Figure 4.2 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.

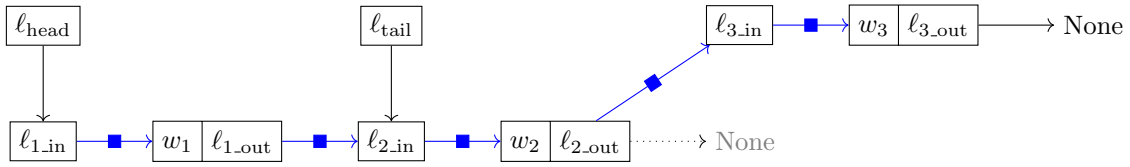
4.2.3 Dequeue

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 head node 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 function returns None. Otherwise, there is a node just after the head node, which is the first node of the queue. To dequeue it, we first read the associated value, and next we swing the head pointer to it, making it the new head node. Finally, we return the value we read.

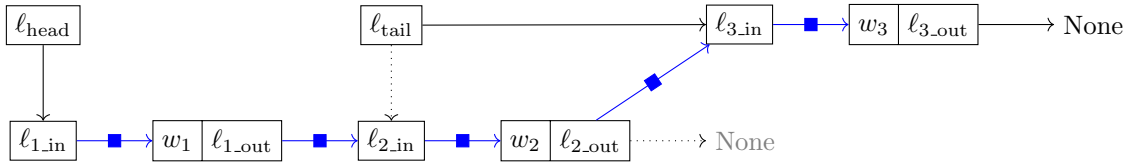
Since all of these operations interact with the queue, we shall only perform them after having acquired h_{lock}.



(a) Queue after creating the new node $(\ell_{3\_\text{in}}, w_3, \ell_{3\_\text{out}})$ 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 4.2: Enqueuing an element to a queue with one element. The illustrations assume that no dequeue is happening, hence ℓ_{head} stays the same.

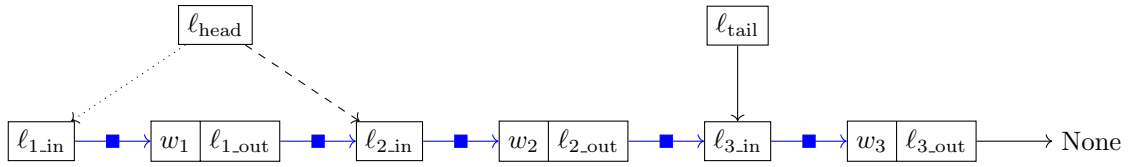


Figure 4.3: Dequeueing an element (w_2) from a queue with two elements (w_2, w_3). The dotted line represents the state before the dequeue, and the dashed line is the state after dequeuing.

Figure 4.3 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 head node is not deleted, it just becomes unreachable from the head pointer. In this way, the linked list only ever grows.

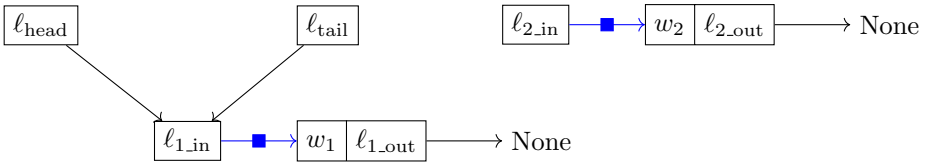
4.2.4 Observations on the Two-Lock Michael-Scott Queue

Now that we have seen the implementation, we point out the following noteworthy observations about the behaviour of the queue.

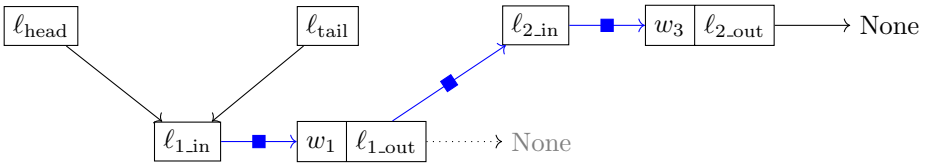
1. The tail node is always either the last or second last node in the linked list.
2. All but the last pointer in the linked list (the pointer to None) never change.
3. Nodes in the linked list are never deleted. Hence, the linked list only ever grows.
4. The tail can lag one node behind the head.
5. 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**)

```
1  initialize ≜
2     let node = ref (None, ref (None)) in
3     let H_lock = newLock() in
4     let T_lock = newLock() in
5     ref ((ref (node), ref (node)), (H_lock, T_lock))
6
7  enqueue Q value ≜
8     let node = ref (Some value, ref (None)) in
9     acquire(snd(snd(! Q)));
10    snd(!(!(snd(fst(! Q))))) ← node;
11    snd(fst(! Q)) ← node;
12    release(snd(snd(! Q)))
13
14 dequeue Q ≜
15    acquire(fst(snd(! Q)));
16    let node = !(fst(fst(! Q))) in
17    let new_head = !(snd(! node)) in
18    if new_head = None then
19       release(fst(snd(! Q)));
20       None
21    else
22       let value = fst(! new_head) in
23       fst(fst(! Q)) ← new_head;
24       release(fst(snd(! Q)));
25       value
```

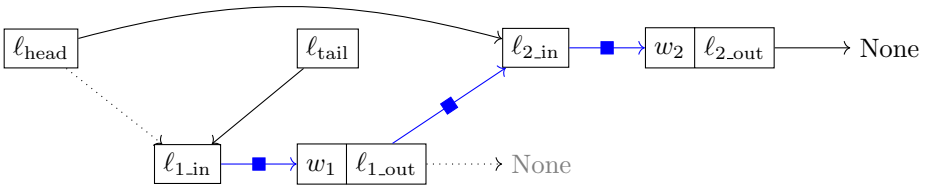
Figure 4.4: Implementation of Two-Lock M&S Queue in HeapLang.



(a) The queue is initially empty, and an enqueueing thread has created a node $(\ell_{2\_\text{in}}, w_2, \ell_{2\_\text{out}})$ that it wishes to append to the linked list.



(b) The thread executes line 10 which adds the created node to the linked list.



(c) Before the thread executes line 11 to swing the tail pointer ℓ_{tail}, another thread dequeues the node that was just enqueued, which swings the head pointer, ℓ_{head} to it. The tail is now lagging behind.

Figure 4.5: Illustrations of a scenario that makes the head lag behind the tail.

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

Observation 1 captures the fact that, while enqueueing, a new node is first added to the linked list, and then later the tail pointer 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 pointer will only ever point to the last or second last node. 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 2 means that we can mark all pointers in the queue, except the pointer to None, as persistent.

In the original implementation [Michael and Scott, 1996], nodes are freed after being dequeued. However, since our language, HeapLang, is a garbage-collected language, we do not explicitly free the nodes. This then leads to observation 3. From our perspective, the linked list only grows.

Observation 4 might seem a little surprising, and indeed it stands in contrast to property 5 in Michael and Scott [1996], which states "Tail always points to a node in the linked list, because it never lags behind Head, so it can never point to a deleted node.". 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 5.3.4). 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 one node behind the head. Figure 4.5 illustrates such a program trace.

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 5 is a simple consequence of the implementation using two locks.

Chapter 5

Proving Specifications for the Two-Lock Michael-Scott Queue

5.1 Introduction

In this chapter we show that the Two-Lock M&S Queue satisfies the three queue specifications introduces in chapter 3. Section 3.5 of said chapter showed us that we get the sequential and concurrent specifications essentially for free, granted we can prove that Two-Lock M&S Queue satisfies the Hocap-style specification. However, for instructive purposes, we will in this chapter give the sequential and concurrent specifications directly, as the proof of the Hocap-style specification can get somewhat overwhelming.

In section 5.2 we prove that the Two-Lock M&S Queuesatisfies the sequential specification. This also introduces some of the basic predicates we use for the queue. Next, in section 5.3, we prove the concurrent specification, which introduces some more complexities, such as invariant, and shows us how we can work with concurrent programs. Finally, section 5.4 shows how to modify the proof of the concurrent specification to prove the Hocap style specification.

5.2 Sequential Specification

5.2.1 Sequential Queue Predicate

Since the queue uses two locks, we will need two ghost names; one for each lock (cf. section 2.2.6). We thus let $\mathit{SeqQgnames}$ be the set of pairs containing ghost names. For an element $G \in \mathit{SeqQgnames}$, the first element of the pair, written $G.\gamma_{\text{Hlock}}$, will contain the ghost name for the head lock, and the second element, $G.\gamma_{\text{Tlock}}$, the ghost name for the tail lock.

To prove the specification, we must give a specific isQueue_S predicate. With the points we discussed in section 4.2.4 in mind, we give our definition of the queue predicate for the sequential specification and explain it afterwards.

Definition 5.2.1 (Two-Lock M&S Queue - isQueue_S Predicate)**.**

$$
\begin{aligned}
\text{isQueue}_\text{S}(v_q, xs_v, G) \triangleq \exists \ell_{\text{queue}}, \ell_{\text{head}}, \ell_{\text{tail}} \in \mathit{Loc}.\ \exists h_{\text{lock}}, t_{\text{lock}} \in \mathit{Val}.\\
& v_q = \ell_{\text{queue}} * \ell_{\text{queue}} \mapsto^{\square} ((\ell_{\text{head}}, \ell_{\text{tail}}), (h_{\text{lock}}, t_{\text{lock}})) *\\
& \exists xs_{\text{queue}} \in \mathit{List}\ (\mathit{Loc} \times \mathit{Val} \times \mathit{Loc}).\ \exists x_{\text{head}}, x_{\text{tail}} \in (\mathit{Loc} \times \mathit{Val} \times \mathit{Loc}).\\
& \text{projVal}(xs_{\text{queue}}) = \text{wrapSome}(xs_v) *\\
& \text{isLL}(xs_{\text{queue}} \mathbin{+\!\!+} [x_{\text{head}}]) *\\
& \ell_{\text{head}} \mapsto \text{in}(x_{\text{head}}) *\\
& \ell_{\text{tail}} \mapsto \text{in}(x_{\text{tail}}) * \text{isLast}(x_{\text{tail}}, (xs_{\text{queue}} \mathbin{+\!\!+} [x_{\text{head}}])) *\\
& \text{isLock}(G.\gamma_{\text{Hlock}}, h_{\text{lock}}, \mathsf{True}) *\\
& \text{isLock}(G.\gamma_{\text{Tlock}}, t_{\text{lock}}, \mathsf{True})
\end{aligned}
$$

This isQueue_S predicate states that the value v_q is a location, which persistently points to the structure containing the head and tail pointers, and the two locks. It also relates the abstract state xs_v to the concrete state by stating that if you strip away the locations in xs_{queue} (using projVal; see appendix A.1.4) and wrap the values in the abstract state xs_v in Some (using wrapSome; see appendix A.1.5), 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 $\text{isLast}(x, xs)$ simply asserts the existence of some xs', so that $xs = x :: xs'$ (defined formally in Appendix A.1.2).

Next, we have the isLock predicate for our two locks. Since we are in a sequential setting, the locks are superfluous, hence they simply protect True.

The isLL predicate essentially creates the structure seen in the examples of section 4.2. 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 $\text{isLL\_chain}(xs)$ persistent for all xs.

Definition 5.2.2 (Linked List Chain Predicate)**.**

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

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

Definition 5.2.3 (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 4.2c, we would use the list $xs = [(\ell_{3\_\text{in}}, w_3, \ell_{3\_\text{out}}); (\ell_{2\_\text{in}}, w_2, \ell_{2\_\text{out}}); (\ell_{1\_\text{in}}, w_1, \ell_{1\_\text{out}})]$. $\text{isLL}(xs)$ will expand to $\ell_{3\_\text{out}} \mapsto \text{None} * \text{isLL\_chain}(xs)$, and $\text{isLL\_chain}(xs)$ expands to

$$\begin{aligned}
&\ell_{3_\text{in}} \mapsto^{\square} (w_3, \ell_{3_\text{out}}) * \ell_{2_\text{out}} \mapsto^{\square} \ell_{3_\text{in}} * \\
&\ell_{2_\text{in}} \mapsto^{\square} (w_2, \ell_{2_\text{out}}) * \ell_{1_\text{out}} \mapsto^{\square} \ell_{2_\text{in}} * \\
&\ell_{1_\text{in}} \mapsto^{\square} (w_1, \ell_{1_\text{out}})
\end{aligned}$$

Note how this matches the structure of the linked list in figure 4.2c.

The isLL predicate turns out to be quite fundamental in describing both the Two-Lock M&S Queue and the Lock-Free M&S Queue, and we shall generally have such a predicate for both of these when we prove that they satisfy the specifications. These proofs require us to manipulate specific isLL propositions quite a bit in conjunction with the queue changing – appendix A.2 shows the specific lemmas we use, but the proof outlines will generally not mention the lemmas explicitly.

5.2.2 Proof Outline

Initialise

Lemma 4 (Two-Lock M&S Queue Sequential Specification - Initialise)**.**

$$\{\mathsf{True}\}\ \text{initialize}\ ()\ \{v_q.\exists G.\ \text{isQueue}_\text{S}(v_q, [], G)\}$$

Proof. Proving the initialise specification amounts to stepping through the code, giving us the required resources, and then using these to create an instance of isQueue_S with the obtained resources. To begin with, we step through line 2 which creates the first node $x_1 = (\ell_{1\_\text{in}}, \text{None}, \ell_{1\_\text{out}})$ with $\ell_{1\_\text{out}} \mapsto \text{None}$ and $\ell_{1\_\text{in}} \mapsto (\text{None}, \ell_{1\_\text{out}})$. We can then update the latter points-to proposition to become persistent, giving us $\ell_{1\_\text{in}} \mapsto^{\square} (\text{None}, \ell_{1\_\text{out}})$. Next, on lines 3 and 4, we create the two locks. We use the specification for newLock for each of its invocations. Both times, we specify that the lock should protect True. This

gives us two ghost names, γ_{Hlock}, γ_{Tlock}, which we will collect in a *SeqQgnames* pair, G. Finally, we step through the allocations of the head, tail, and queue pointers on line 5. This 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 and tail pointers, and the 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. 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

Lemma 5 (Two-Lock M&S Queue Sequential Specification - Enqueue)**.**

$$\forall v_q, v, xs_v, G.\ \{\text{isQueue}_{\text{S}}(v_q, xs_v, G)\}\ \text{enqueue}\ \ v_q\ \ v\ \{w.\ \text{isQueue}_{\text{S}}(v_q, (v :: xs_v), G)\}$$

Proof. We assume the pre-condition $\text{isQueue}_{\text{S}}(v_q, xs_v, G)$ which tells us that v_q is some location ℓ_{queue}, and in particular, we have following:

$$\ell_{\text{queue}} \mapsto^{\square} ((\ell_{\text{head}}, \ell_{\text{tail}}), (h_{\text{lock}}, t_{\text{lock}})) \tag{5.1}$$
$$\text{projVal}(xs_{\text{queue}}) = \text{wrapSome}(xs_v) \tag{5.2}$$
$$x_{\text{tail}} \mapsto \text{None} * \text{isLL\_chain}(xs_{\text{queue}} +\!\!+ [x_{\text{head}}]) \tag{5.3}$$
$$\ell_{\text{tail}} \mapsto \text{in}(x_{\text{tail}}) * \text{isLast}(x_{\text{tail}}, (xs_{\text{queue}} +\!\!+ [x_{\text{head}}])) \tag{5.4}$$
$$\text{isLock}(G.\gamma_{\text{Tlock}}, t_{\text{lock}}, \mathsf{True}) \tag{5.5}$$

We proceed to step into the enqueue function. Firstly, on line 8, we create a new node x_{new}, with $\text{val}(x_{\text{new}}) = \text{Some}\, v$. Next, on line 9 we acquire the tail lock. We step through the dereference and the projections using 5.1. To acquire the lock we use the acquire specification with 5.5. This gives us $\text{locked}(Qg.\gamma_{\text{Tlock}})$ and True.

Line 10 adds node x_{new} to the linked list. We first use 5.1 to step to the dereference of ℓ_{tail}. From 5.4, we know the dereference results in $\text{in}(x_{\text{tail}})$. Because x_{tail} is in the linked list, we can use 5.3 to conclude that x_{tail} is a node, and hence we can perform the load and the projection to get to the final store operation: $\text{out}(x_{\text{tail}}) \leftarrow \text{in}(x_{\text{new}})$. Using the points-to proposition from 5.3, we perform the store, which in turn gives us $\text{out}(x_{\text{tail}}) \mapsto \text{in}(x_{\text{new}})$. We make this persistent and combine it with the isLL_chain proposition from 5.3 to conclude $\text{isLL}(x_{\text{new}} :: xs_{\text{queue}} +\!\!+ [x_{\text{head}}])$.

The next line (line 11) swings the tail pointer to x_{new}. 5.4 and 5.4 gives us all the required resources to step through the code and perform the store. Afterwards, we get $\ell_{\text{tail}} \mapsto \text{in}(x_{\text{new}})$.

Finally, we get to line 12 which releases the lock. We use the specification for release, giving up $\text{locked}(Qg.\gamma_{\text{Tlock}})$. The only thing left is to prove the postcondition: $\text{isQueue}_{\text{S}}(v_q, (v :: xs_v), G)$. For the existentials, we pick $x_{\text{new}} :: xs_{\text{queue}}$ as the queue, and as the tail node, we chose x_{new}. The remaining choices are similar to those we got from the pre-condition. With these choices, the remaining proof obligations become straightforward; we already have the required pointers and the isLL proposition. The relationship between the abstract and concrete states follows from $\text{val}(x_{\text{new}}) = \text{Some}\, v$ and 5.2. □

Dequeue

Lemma 6 (Two-Lock M&S Queue Sequential Specification - Dequeue)**.**

$$\forall v_q, xs_v, G.\ \{\text{isQueue}_{\text{S}}(v_q, xs_v, G)\}$$
$$\text{dequeue}\ \ v_q$$
$$\left\{w.\ \begin{array}{l}(xs_v = [] * w = \text{None} * \text{isQueue}_{\text{S}}(v_q, xs_v, G)) \vee \\ (\exists v, xs'_v.\ xs_v = xs'_v +\!\!+ [v] * w = \text{Some}\, v * \text{isQueue}_{\text{S}}(v_q, xs'_v, G))\end{array}\right\}$$

Proof. We assume the pre-condition $\text{isQueue}_{\text{S}}(v_q, xs_v, G)$ which gives us that v_q is some location ℓ_{queue},

and the following propositions.

$$\ell_{\text{queue}} \mapsto^{\square} ((\ell_{\text{head}}, \ell_{\text{tail}}), (h_{\text{lock}}, t_{\text{lock}})) \tag{5.6}$$
$$\text{projVal}(xs_{\text{queue}}) = \text{wrapSome}(xs_v) \tag{5.7}$$
$$x_{\text{tail}} \mapsto \text{None} * \text{isLL\_chain}(xs_{\text{queue}} \mathbin{+\!\!+} [x_{\text{head}}]) \tag{5.8}$$
$$\ell_{\text{head}} \mapsto \text{in}(x_{\text{head}}) * \tag{5.9}$$
$$\text{isLock}(G.\gamma_{\text{Hlock}}, h_{\text{lock}}, \mathsf{True}) \tag{5.10}$$

We perform the function application and step into the function. We first do the superfluous acquire on line 15 using the acquire specification and 5.10. This gives us locked($G.\gamma_{\text{Hlock}}$).
Next we step to line 16 which dereferences the head node. We perform the loads and projections using 5.6 and 5.9, which tells us that the *node* variable becomes in(x_{head}). From 5.8, we know that x_{head} is in the linked list, hence it must be a node.
We step to line 17 which finds out what x_{head} points to. As x_{head} is a node, we can step to the load: !(out(x_{head})). The result of this load is either None or another node $x_{\text{head\_next}}$, depending on whether or not xs_{queue} is empty. So we consider both cases.

Case xs_{queue} **is empty**: In this case, 5.8 simply asserts isLL[x_{head}], which by definition tells us that $x_{\text{head}} \mapsto$ None. Hence, the "if" on line 18 will take the "then" branch, so we step to line 19. Here we release the lock, giving up locked($G.\gamma_{\text{Hlock}}$), and return None on the next line. What remains is to prove the post-condition with $w =$ None. We can easily do this by proving the first disjunction. From 5.7 with the fact that $xs_{\text{queue}} = []$ we can conclude that xs_v is empty, and since we haven't modified the queue, we can prove isQueue$_{\text{S}}(v_q, xs_v, G)$ using the same resources we got from the pre-condition.

Case xs_{queue} **is not empty**: In this case, we can conclude that there must be some node $x_{\text{head\_next}}$, which is the first node in xs_{queue}. I.e.

$$xs_{\text{queue}} = xs'_{\text{queue}} \mathbin{+\!\!+} [x_{\text{head\_next}}] \tag{5.11}$$

We can thus use the isLL predicate to conclude that x_{head} must point to $x_{\text{head\_next}}$. Hence the "else" branch will be taken, so we step to line 25. Since $x_{\text{head\_next}}$ is part of the linked list, it must be a node, allowing us to extract its value val($x_{\text{head\_next}}$) in the first line of the else branch.
We step to line 23 which swings the head pointer, ℓ_{head} to $x_{\text{head\_next}}$. We perform these steps using 5.6 and 5.9, which then gives us $\ell_{\text{head}} \mapsto \text{in}(x_{\text{head\_next}})$.
Finally, we release the lock on line 24, giving up locked($G.\gamma_{\text{Hlock}}$) and return the value: val($x_{\text{head\_next}}$).
We must now prove the post-condition with $w =$ val($x_{\text{head\_next}}$), and this time we prove the second disjunct. From 5.7 and 5.11 we can deduce

$$xs_v = xs'_v \mathbin{+\!\!+} [v] \tag{5.12}$$
$$\text{projVal}(xs'_{\text{queue}}) = \text{wrapSome}(xs'_v) \tag{5.13}$$
$$\text{val}(x_{\text{head\_next}}) = \text{Some}\, v \tag{5.14}$$

5.12 and 5.14 proves the first part of the post-condition. What remains is to show isQueue$_{\text{S}}(v_q, xs'_v, G)$. For the existentials, we pick xs'_{queue} as the queue and $x_{\text{head\_next}}$ as the head node. The rest are similar to the variables we got from the pre-condition. With these choices, we can prove the predicate using the resources we have established.

□

5.3 Concurrent Specification

5.3.1 Concurrent Queue Predicate

For the concurrent specification, we need to allow for multiple threads to access the queue resources to perform the queue operations, as showcased in the sequential case. The concurrent specification

enforces this possibility by asserting that the queue predicate, isQueue_C, is persistent, hence duplicable. The resources needed by the queue, however, are not persistent. The solution is to collect the required resources in an *invariant* which *is* persistent and can hence be given to multiple threads. The invariant we define here additionally describes the concrete state of the queue. In the proofs of the queue functions, we shall then access the required resources through the invariant. We now present the invariant and explain it afterwards.

Definition 5.3.1 (Two-Lock M&S Queue Concurrent Invariant)**.**

$$
\begin{array}{ll}
\text{I}_{\text{TLC}}(\Psi, \ell_{\text{head}}, \ell_{\text{tail}}, G) \triangleq & \\
\exists xs_v.\ \text{All}(xs_v, \Psi) * & \text{(abstract state)} \\
\exists xs, xs_{\text{queue}}, xs_{\text{old}}, x_{\text{head}}, x_{\text{tail}}. & \text{(concrete state)} \\
xs = xs_{\text{queue}} \mathbin{+\!\!+} [x_{\text{head}}] \mathbin{+\!\!+} xs_{\text{old}} * & \\
\text{isLL}(xs) * & \\
\text{projVal}(xs_{\text{queue}}) = \text{wrapSome}(xs_v) * & \\
(& \\
\quad \ell_{\text{head}} \mapsto \text{in}(x_{\text{head}}) * \ell_{\text{tail}} \mapsto \text{in}(x_{\text{tail}}) * \text{isLast}(x_{\text{tail}}, xs) * & \textbf{(Static)} \\
\quad \text{TokNE}\ G * \text{TokND}\ G * \text{TokUpdated}\ G & \\
\vee & \\
\quad \ell_{\text{head}} \mapsto \text{in}(x_{\text{head}}) * \ell_{\text{tail}} \mapsto^{\frac{1}{2}} \text{in}(x_{\text{tail}}) * & \textbf{(Enqueue)} \\
\quad (\text{isLast}(x_{\text{tail}}, xs) * \text{TokBefore}\ G \vee \text{isSndLast}(x_{\text{tail}}, xs) * \text{TokAfter}\ G) * & \\
\quad \text{TokE}\ G * \text{TokND}\ G & \\
\vee & \\
\quad \ell_{\text{head}} \mapsto^{\frac{1}{2}} \text{in}(x_{\text{head}}) * \ell_{\text{tail}} \mapsto \text{in}(x_{\text{tail}}) * \text{isLast}(x_{\text{tail}}, xs) * & \textbf{(Dequeue)} \\
\quad \text{TokNE}\ G * \text{TokD}\ G * \text{TokUpdated}\ G & \\
\vee & \\
\quad \ell_{\text{head}} \mapsto^{\frac{1}{2}} \text{in}(x_{\text{head}}) * \ell_{\text{tail}} \mapsto^{\frac{1}{2}} \text{in}(x_{\text{tail}}) * & \textbf{(Both)} \\
\quad (\text{isLast}(x_{\text{tail}}, xs) * \text{TokBefore}\ G \vee \text{isSndLast}(x_{\text{tail}}, xs) * \text{TokAfter}\ G) * & \\
\quad \text{TokE}\ G * \text{TokD}\ G & \\
) &
\end{array}
$$

In contrast to the sequential specification, the abstract state is now existentially quantified, hence the exact contents of the queue are not tracked. Instead, we have added the proposition $\text{All}(xs_v, \Psi)$, which states that all values in xs_v (i.e. the values currently in the queue) satisfy the predicate Ψ (see appendix A.1.6 for formal definition). This will allow us to conclude that dequeued values satisfy Ψ.
The concrete state of the queue is still reflected in the abstract state through projecting out the values of the nodes (via projVal), and wrapping the values in the queue in Some (via wrapSome). 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 4.
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 5. 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 (as shown in the sequential specification), 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 $\ell_{\text{tail}} \mapsto^{\frac{1}{2}} \text{in}(x_{\text{tail}})$ from an earlier access, and it will obtain the existence of some new x'_{tail}, such that $\ell_{\text{tail}} \mapsto^{\frac{1}{2}} \text{in}(x'_{\text{tail}})$. Combining the two points-to

predicates allows us to conclude that $\text{in}(x_{\text{tail}}) = \text{in}(x'_{\text{tail}})$ and using lemma 33 we can further conclude $x_{\text{tail}} = x'_{\text{tail}}$. 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 1: 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; see appendix A.1.3).
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*. Importantly, tokens are non-duplicable and owning two of the same token allows us to deduce False. 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. To distinguish between different tokens, we use ghost names. Each token is associated with a ghost name, γ, and we write $\text{Token}(\gamma)$ for said token. We refer to section 8.1 for a presentation of the resource algebra used to define tokens. The following two lemmas shows us that we can create new tokens and that they are exclusive, as explained above.

Lemma 7 (Token Alloc)**.** $\vdash \Rrightarrow \exists \gamma.\ \text{Token}(\gamma)$

Lemma 8 (Token Exclusive)**.** $\text{Token}(\gamma) * \text{Token}(\gamma) \vdash \mathsf{False}$

We will use several tokens, each associated with their own ghost name. We collect all these new ghost names in *ConcQgnames*, which then contains the ghost names for the two locks and the ghost names for the tokens.
We introduce some notation to refer to specific tokens related to a tuple G. For example, if we wish to refer to the token associate with $G.\gamma_E$, we write TokE G, which projects out γ_E, and asserts ownership of the token associated with it. That is, TokE $G = \text{Token}(G.\gamma_E)$. We proceed to explain the meaning of each of the tokens used in the invariant.

- TokNE G represents that no threads are enqueueing.
- TokE G represents that a thread is enqueueing.
- TokND G represents that no threads are dequeueing.
- TokD G represents that a thread is dequeueing.
- TokBefore G represents that an enqueueing thread has not yet added the new node to the linked list.
- TokAfter G represents that an enqueueing thread has added the new node to the linked list, but not yet swung the tail.
- TokUpdated G is defined as TokBefore $G *$ TokAfter G, and represents that the queue is up to date.

We note that the concurrent specification for the Two-Lock M&S Queue *can* be proven using queue invariant 5.3.1 directly, 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 5.3.1 and has the benefit of being easier to work with in the mechanised proofs. Thus, in the mechanised proofs, we always rewrite to the simpler invariant when opening the invariant. We define it as follows.

Definition 5.3.2 (Simplified Two-Lock M&S Queue Invariant)**.**

$$
\begin{aligned}
&\mathrm{I}'_{\mathrm{TLC}}(\Psi, \ell_{\mathrm{head}}, \ell_{\mathrm{tail}}, G) \triangleq \\
&\exists xs_v.\ \mathrm{All}(xs_v, \Psi) * && \text{(abstract state)} \\
&\exists xs, xs_{\mathrm{queue}}, xs_{\mathrm{old}}, x_{\mathrm{head}}, x_{\mathrm{tail}}. && \text{(concrete state)} \\
&xs = xs_{\mathrm{queue}} \mathbin{+\!\!+} [x_{\mathrm{head}}] \mathbin{+\!\!+} xs_{\mathrm{old}} * \\
&\mathrm{isLL}(xs) * \\
&\mathrm{projVal}(xs_{\mathrm{queue}}) = \mathrm{wrapSome}(xs_v) * \\
&((\ell_{\mathrm{head}} \mapsto \mathrm{in}(x_{\mathrm{head}}) * \mathrm{TokND}\ G) \vee (\ell_{\mathrm{head}} \mapsto^{\frac{1}{2}} \mathrm{in}(x_{\mathrm{head}}) * \mathrm{TokD}\ G)) * \\
&(\\
&\quad (\ell_{\mathrm{tail}} \mapsto \mathrm{in}(x_{\mathrm{tail}}) * \mathrm{isLast}(x_{\mathrm{tail}}, xs) * \mathrm{TokNE}\ G * \mathrm{TokUpdated}\ G) \vee \\
&\quad (\\
&\qquad \ell_{\mathrm{tail}} \mapsto^{\frac{1}{2}} \mathrm{in}(x_{\mathrm{tail}}) * \mathrm{TokE}\ G * \\
&\qquad ((\mathrm{isLast}(x_{\mathrm{tail}}, xs) * \mathrm{TokBefore}\ G) \vee (\mathrm{isSndLast}(x_{\mathrm{tail}}, xs) * \mathrm{TokAfter}\ G)) \\
&\quad) \\
&)
\end{aligned}
$$

As is evident, it contains the same propositions as the original invariant, but simply in a different structure. This structure allows for more linear reasoning; the common propositions between the different queue states are now "grouped" together.

With this, we now give our definition of the queue predicate, $\mathrm{isQueue_C}$. In the below, we let $\mathcal{N}$ be some namespace.

Definition 5.3.3 (Two-Lock M&S Queue - $\mathrm{isQueue_C}$ Predicate)**.**

$$
\begin{aligned}
\mathrm{isQueue_C}(\Psi, v_q, G) \triangleq &\exists \ell_{\mathrm{queue}}, \ell_{\mathrm{head}}, \ell_{\mathrm{tail}} \in \mathit{Loc}.\ \exists h_{\mathrm{lock}}, t_{\mathrm{lock}} \in \mathit{Val}. \\
&v_q = \ell_{\mathrm{queue}} * \ell_{\mathrm{queue}} \mapsto^{\square} ((\ell_{\mathrm{head}}, \ell_{\mathrm{tail}}), (h_{\mathrm{lock}}, t_{\mathrm{lock}})) * \\
&\boxed{\mathrm{I}_{\mathrm{TLC}}(\Psi, \ell_{\mathrm{head}}, \ell_{\mathrm{tail}}, G)}^{\mathcal{N}.queue} * \\
&\mathrm{isLock}(G.\gamma_{\mathrm{Hlock}}, h_{\mathrm{lock}}, \mathrm{TokD}\ G) * \\
&\mathrm{isLock}(G.\gamma_{\mathrm{Tlock}}, t_{\mathrm{lock}}, \mathrm{TokE}\ G).
\end{aligned}
$$

In contrast to our definition of $\mathrm{isQueue_S}$, the locks now protect TokE G and TokD G. The idea is that, when an enqueueing thread obtains t_{lock}, they will obtain the TokE G token, which allows them to conclude that the queue state is either **Static** or **Dequeue**. Similarly for a dequeueing thread.

5.3.2 Linearisation Points

An important notion for concurrent algorithms is *linearisability* [Herlihy and Wing, 1990]. Linearisability is a non-blocking property[1] that helps reason about which behaviours are possible. Both versions of the M&S Queue are linearisable, which rules out undesired behaviours.

One way to characterise linearisability is through the concept of *linearisation points*. We say that a function (such as enqueue or dequeue) is linearisable, if for all invocations of the function, there is a specific point in time between the invocation and the response where the effect of the function appears to takes place; before that point, the effect hasn't taken place, and nothing needs to be done afterwards to complete the effect. We call such a point a linearisation point.

For example, enqueue has a single linearisation point at the instruction that appends the newly created node to the linked list (the store on line 10). At exactly that point, the effect of the enqueue takes place. dequeue is slightly more complicated as it has multiple linearisation points. If the queue is empty when we read the head node's out pointer on line 17, then at that read, the dequeue function is guaranteed to return None, interpreting the queue as empty. In this case the dequeue doesn't change the

[1] Other properties, such as serialisability, inherently require implementations to be blocking. Linearisability does not require this, but linearisable algorithms can of course still be blocking.

queue but merely observes it, and the observation happens exactly at the read of the head node's out pointer, making it the linearisation point.
On the hand, if the queue was not empty at that point, then the linearisation point is at line 23, when we swing the head pointer. At that very store operation, the effect of dequeue occurs.

Linearisation points are closely tied to updates of the abstract state of the queue. The abstract state of the queue changes only at linearisation points. This is consistent with the notion that the effects of the function takes effect at the linearisation points – updates to the abstract state happen atomically as we would intuitively want it to. This link to the abstract state becomes even more prevalent when we prove Hocap-style specifications in sections 5.4 and section 7.5.

5.3.3 Proof Outline

Firstly, we must show that isQueue_C 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 of the three specifications largely have the same structure as 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

Lemma 9 (Two-Lock M&S Queue Concurrent Specification - Initialise)**.**

$$\{\mathsf{True}\}\ \text{initialize}\ ()\ \{v_q.\exists G.\ \text{isQueue}_\text{C}(\Psi, v_q, G)\}$$

Proof. We first step through line 2 which gives us the head node of the linked list. Following this, we must create the two locks. As the locks had to protect tokens, we must first create these. To create the two tokens, we use lemma 7 twice, which gives us two ghost names, one for each of the tokens. We put the ghost names into a tuple G, and write TokE G and TokD G for the two ghost resources created by the lemma. We then use the newLock specification to step through the code and create the locks, giving up the two tokens. Next, we step to line 5 and create the ℓ_queue, ℓ_head, and ℓ_tail pointers with $\ell_\text{queue} \mapsto ((\ell_\text{head}, \ell_\text{tail}), (h_\text{lock}, t_\text{lock}))$. We make this persistent.
All that remains then is to prove the postcondition; the isQueue_C predicate. We have both the points-to predicate of ℓ_queue and the two isLock predicates (from the newLock specification). So all that remains is the invariant.
We create I_TLC 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 G, TokND G, TokBefore G, and TokAfter G, again using lemma 7. We combine TokBefore G and TokAfter G to get TokUpdated G, and we now have all the tokens we need to create I_TLC in the **Static** state. To create the invariant from I_TLC, we use the Inv-alloc rule. □

Enqueue

Lemma 10 (Two-Lock M&S Queue Concurrent Specification - Enqueue)**.**

$$\forall v_q, v, G.\ \{\text{isQueue}_\text{C}(\Psi, v_q, G) * \Psi(v)\}\ \text{enqueue}\ v_q\ v\ \{w.\mathsf{True}\}$$

Proof. We assume the pre-condition, which tells us that v_q is a location ℓ_queue, and we have:

$$\Psi(v) \tag{5.15}$$
$$\ell_\text{queue} \mapsto^{\square} ((\ell_\text{head}, \ell_\text{tail}), (h_\text{lock}, t_\text{lock})) \tag{5.16}$$
$$\boxed{\text{I}_\text{TLC}(\Psi, \ell_\text{head}, \ell_\text{tail}, G)}^{\mathcal{N}.queue} \tag{5.17}$$
$$\text{isLock}(G.\gamma_\text{Tlock}, t_\text{lock}, \text{TokE}\ G) \tag{5.18}$$

We perform the function application and step into enqueue to line 8. We create the new node as before, giving us x_new, with $\text{val}(x_\text{new}) = \text{Some}\, v$.
On line 9 we then use the acquire specification with 5.18 to acquire $\text{locked}(G.\gamma_\text{Tlock})$ and TokE G.

Following this, we step to line 10. Using 5.16 we can step to the load of ℓ_{tail}. This is where we meet our first challenge; in order to perform the store, we must open the invariant to access the points-to predicate regarding ℓ_{tail}.
As invariants can only be opened if the expression being considered is atomic, we use a bind rule to "focus" on the load of ℓ_{tail}. We proceed to open the invariant, and since we have TokE G, we know that the queue is in state **Static** or **Dequeue**. In any case, we get that $\ell_{\text{tail}} \mapsto \text{in}(x_{\text{tail}})$, for some x_{tail} which is the last node in the linked list. As x_{tail} is in the linked list, we further deduce that it is a node.
We can now perform the load of ℓ_{tail} which results in $\text{in}(x_{\text{tail}})$. We must now close the invariant. We split up the points-to predicate $\ell_{\text{tail}} \mapsto \text{in}(x_{\text{tail}})$ in two, which leaves us with two of $\ell_{\text{tail}} \mapsto^{\frac{1}{2}} \text{in}(x_{\text{tail}})$. 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 G, but we gain TokNE G and TokAfter G.
As x_{tail} was a node we can step to $\text{out}(x_{\text{tail}}) \leftarrow \text{in}(x_{\text{new}})$. However, the points-to predicate concerning $\text{out}(x_{\text{tail}})$ isn't persistent, and is hence inside the invariant. We thus have to open the invariant again. Since we have TokNE G and TokAfter G, we know that we are in the before case of either state **Enqueue** or **Both**.
The invariant hence gives us $\ell_{\text{tail}} \mapsto^{\frac{1}{2}} \text{in}(x'_{\text{tail}})$ for some x'_{tail}. However, since we kept $\ell_{\text{tail}} \mapsto^{\frac{1}{2}} \text{in}(x_{\text{tail}})$, we can combine these, allowing us to conclude that $\text{in}(x_{\text{tail}}) = \text{in}(x'_{\text{tail}})$. As both x_{tail} and x_{tail} are nodes, we apply 33 to conclude $x_{\text{tail}} = x'_{\text{tail}}$. This now gives us that $\text{out}(x_{\text{tail}}) \mapsto \text{None}$, allowing us to perform the store. Thus, x_{new} is added to the linked list, and this is a linearisation point. As such we must update the abstract state to reflect the change. We do this by closing the invariant with $(v :: xs_v)$ as the abstract state, where xs_v is the abstract state we got when we opened the invariant. Note that we have $\Psi(v)$ (from 5.15), hence we will be able to conclude $\text{All}((v :: xs_v), \Psi)$. For the concrete state, we pick $x_{\text{new}} :: xs$, where xs is the concrete state we got when we opened the invariant. We close the invariant in the after case of either state **Enqueue** or **Both**, giving up TokAfter G, and obtaining TokBefore G.

We step to line 11, which swings the tail pointer to x_{new}. Using 5.16 we step to $\ell_{\text{tail}} \leftarrow \text{in}(x_{\text{new}})$. However, to perform this store we must first know that ℓ_{tail} points to something. This resource is inside the invariant, so we open the invariant one last time. Due to having TokNE G and TokBefore G, we know that we are in the after case of state **Enqueue** or **Both**. This time we get $\ell_{\text{tail}} \mapsto^{\frac{1}{2}} x''_{\text{tail}}$ for some x''_{tail}, where x''_{tail} is 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 our half of the points-to predicate of ℓ_{tail} (i.e. $\ell_{\text{tail}} \mapsto^{\frac{1}{2}} \text{in}(x_{\text{tail}})$) to get that $x''_{\text{tail}} = x_{\text{tail}}$. Since x_{tail} points to x_{new}, and x''_{tail} points to x'_{new}, we can further conclude that $x_{\text{new}} = x'_{\text{new}}$. 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** or **Dequeue**, giving up TokNE G and TokUpdated G, and getting TokE G.
Finally, on line 12, we release the lock which we can do since we have TokE G and $\text{locked}(G.\gamma_{\text{Tlock}})$. The postcondition merely asserts True, so there is nothing left to prove. □

Dequeue

Lemma 11 (Two-Lock M&S Queue Concurrent Specification - Dequeue)**.**

$$\forall v_q, G.\, \{\text{isQueue}_{\text{C}}(\Psi, v_q, G)\}\ \text{dequeue}\ \ v_q\, \{w. w = \text{None} \vee (\exists v.\, w = \text{Some}\, v * \Psi(v))\}$$

Proof. As usual, we assume the pre-condition giving us that v_q is a location ℓ_{queue} with

$$\ell_{\text{queue}} \mapsto^{\square} ((\ell_{\text{head}}, \ell_{\text{tail}}), (h_{\text{lock}}, t_{\text{lock}})) \tag{5.19}$$

$$\boxed{\text{I}_{\text{TLC}}(\Psi, \ell_{\text{head}}, \ell_{\text{tail}}, G)}^{\mathcal{N}.queue} \tag{5.20}$$

$$\text{isLock}(G.\gamma_{\text{Hlock}}, h_{\text{lock}}, \mathsf{True}) \tag{5.21}$$

We do the function application and step into dequeue. First, on line 15, we acquire the lock, which gives us $\text{locked}(G.\gamma_{\text{Hlock}})$ and TokD G.

Next, we step to line 16 and using 5.19 we get to $!(\ell_{\text{head}})$. To perform this load, we must open the invariant. We open it in state **Static** or **Enqueue** (as we have TokD G), which gives us $\ell_{\text{head}} \mapsto \text{in}(x_{\text{head}})$ for some x_{head}. The isLL predicate further tells us that x_{head} is a node: $\text{in}(x_{\text{head}}) \mapsto^{\square} (\text{val}(x_{\text{head}}), \text{out}(x_{\text{head}}))$. We perform the load, and take half of the points-to predicate: $\ell_{\text{head}} \mapsto^{\frac{1}{2}} \text{in}(x_{\text{head}})$.

We use the other half to close the invariant in state **Dequeue** or **Both**, giving up TokD G, but obtaining TokND G.

We proceed to line 17, which finds out what x_{head} is pointing to. As x_{head} is a node we can step to !(out(x_{head})). To perform this dereference, we must open the invariant. As we own TokND G, the queue must be in state **Dequeue** or **Both**. In any case, we get that there is some x'_{head} with $\ell_{\text{head}} \mapsto^{\frac{1}{2}} x'_{\text{head}}$. Using the fractional points-to predicate we kept from earlier and lemma 33, we can conclude that $x'_{\text{head}} = x_{\text{head}}$. We now perform a case analysis on the contents of the queue: xs_{queue}, similarly to the sequential proof.

Case xs_{queue} **is empty**: In this case, we use the isLL predicate to conclude out(x_{head}) $\mapsto$ None. The expression !(out(x_{head})) hence resolves to None. At this point, we know dequeue will decide that the queue is empty, so this is a linearisation point. We close the invariant in state **Static** or **Enqueue**, giving up TokND G and obtaining TokD G.
As *new_head* was set to None, the "if" on line 18 takes the "then" branch, so we step to line 19. We release the lock, giving up TokD G and locked($G.\gamma_{\text{Hlock}}$), and return None. We must now prove the post-condition with $w =$ None. We easily prove the first disjunct.

Case xs_{queue} **is not empty**: As in the sequential proof, we conclude that x_{head} points to $x_{\text{head\_next}}$ for some $x_{\text{head\_next}}$. That is, we have the following:

$$\text{out}(x_{\text{head}}) \mapsto \text{in}(x_{\text{head\_next}}) \tag{5.22}$$
$$\text{in}(x_{\text{head\_next}}) \mapsto^{\square} (\text{val}(x_{\text{head\_next}}), \text{out}(x_{\text{head\_next}})) \tag{5.23}$$

We perform the dereference, which resolves to in($x_{\text{head\_next}}$). We close the invariant in **Dequeue** or **Both**, meaning we still have TokND G and $\ell_{\text{head}} \mapsto^{\frac{1}{2}}$ in(x_{head}).

This time, the "if" will take the else branch, so we step to line 25. Using 5.23, we conclude that the return value will be val($x_{\text{head\_next}}$).
Next, we step to line 23, which swings the head pointer to $x_{\text{head\_next}}$. Using 5.19, we step to $\ell_{\text{head}} \leftarrow \text{in}(x_{\text{head\_next}})$. Performing this store requires a points-to proposition for ℓ_{head}. Hence, we open the invariant in state **Dequeue** or **Both** (since we have TokND G), which gives us the following:

$$\text{All}(xs_v, \Psi) \tag{5.24}$$
$$xs = xs_{\text{queue}} \mathbin{+\!\!+} [x''_{\text{head}}] \mathbin{+\!\!+} xs_{\text{old}} \tag{5.25}$$
$$\text{isLL}(xs) \tag{5.26}$$
$$\text{projVal}(xs_{\text{queue}}) = \text{wrapSome}(xs_v) \tag{5.27}$$
$$\ell_{\text{head}} \mapsto^{\frac{1}{2}} \text{in}(x''_{\text{head}}) \tag{5.28}$$

for some xs, xs_{queue}, x''_{head}, $xsold$, and xs_v. We combine 5.28 with our half of the points-to predicate to conclude $\ell_{\text{head}} \mapsto \text{in}(x_{\text{head}})$ and use lemma 33 to conclude $x''_{\text{head}} = x_{\text{head}}$.
We can now perform the store, swinging the head pointer to $x_{\text{head\_next}}$, which gives us $\ell_{\text{head}} \mapsto \text{in}(x_{\text{head\_next}})$. This is a linearisation point, so we must update the abstract state xs_v, which we got from the invariant opening.
From 5.22 and 5.26, we can deduce that xs_{queue} isn't empty (as otherwise, we would have $x_{\text{head}} \mapsto$ None which contradicts with 5.22), and in fact, its first element must be $x_{\text{head\_next}}$. That is,

$$xs_{\text{queue}} = xs'_{\text{queue}} \mathbin{+\!\!+} [x_{\text{head\_next}}] \tag{5.29}$$

Combining this with 5.27 allows us to give a similar conclusion to the sequential proof: there exists xs'_v and v such that

$$xs_v = xs'_v \mathbin{+\!\!+} [v] \tag{5.30}$$
$$\text{projVal}(xs'_{\text{queue}}) = \text{wrapSome}(xs'_v) \tag{5.31}$$
$$\text{val}(x_{\text{head\_next}}) = \text{Some}\, v \tag{5.32}$$

By 5.24 and 5.30, we deduce All(xs'_v, Ψ) and $\Psi(v)$. We are now finally ready to close the invariant. We use xs'_v for the abstract state, giving up All(xs'_v, Ψ), but allowing us to keep $\Psi(v)$. For

the concrete state, we use the same xs. Note that by combining 5.25 and 5.29, we have that $xs = xs'_{\text{queue}} \mathbin{+\!\!+} [x_{\text{head\_next}}] \mathbin{+\!\!+} (x_{\text{head}} :: xs_{\text{old}})$, which allows us to pick xs'_{queue} as as the queue, $x_{\text{head\_next}}$ as the head node, and $(x_{\text{head}} :: xs_{\text{old}})$ as the old nodes. Since xs hasn't changed, we prove the isLL predicate using 5.26, and the relationship between xs'_v and xs'_{queue} follows from 5.31. Finally, we set the state of the queue to **Static** or **Enqueue**, giving up TokND G and $\ell_{\text{head}} \mapsto \text{in}(x_{\text{head\_next}})$, and obtaining TokD G.

We step to line 24 and release the lock by giving up TokD G and $\text{locked}(G.\gamma_{\text{Hlock}})$. Lastly, we return the dequeued value: $\text{val}(x_{\text{head\_next}})$, meaning we must prove the post-condition with $w = \text{val}(x_{\text{head\_next}})$. Recall that we got to keep $\Psi(v)$, so using 5.32, we can prove the right disjunct.

□

5.3.4 Discussing the need for Old Nodes

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: xs_{old}. This addition manifests in the end of the proof of dequeue. When we open the invariant to swing ℓ_{head} to $x_{\text{head\_next}}$, 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 $x_{\text{head\_next}}$ is the new head node. Because of this, we can supply the same predicate concerning the location of the x_{tail} in the linked list that we got when we opened the invariant to prove either the **Static** state or the **Enqueue** state.

Had we not used an xs_{old} and essentially just "forgotten" old nodes, we couldn't have done this. Say we defined xs as $xs = xs_{\text{queue}} \mathbin{+\!\!+} [x_{\text{head}}]$ 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 $x_{\text{head\_next}}$) is to close the invariant with $xs' = xs_{\text{queue}} = xs'_{\text{queue}} \mathbin{+\!\!+} [x_{\text{head\_next}}]$. However, clearly $xs' \neq xs$, so we cannot supply the same predicate concerning the location of x_{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 $\text{isLast}(x_{\text{tail}}, xs')$ from $\text{isLast}(x_{\text{tail}}, xs)$, 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 $\text{isSndLast}(x_{\text{tail}}, xs')$ from $\text{isSndLast}(x_{\text{tail}}, xs)$. This is however not provable, since $\text{isSndLast}(x_{\text{tail}}, xs)$ allows for the case where xs'_{queue} is empty. In this case $xs' = [x_{\text{head\_next}}]$ which makes it impossible to prove $\text{isSndLast}(x_{\text{tail}}, xs')$, as it is impossible to the be the second last element in a list of size one.

5.4 Hocap-Style Specification

5.4.1 Introduction

When proving the concurrent specification in the previous section, we were quite careful with tracking the state of the queue, and to some extent, even its 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, it should be no surprise that proving the Hocap-style specification will be very similar. This section will hence focus on the parts where the proof of the Hocap-style specification differs from that of the concurrent specification.

5.4.2 Hocap-Style Queue Predicate

Our definition of the queue predicate, isQueue, is *almost* the same as $\text{isQueue}_{\text{C}}$. First and foremost, the collection of ghost names, *Qgnames*, now contains the additional ghost name: γ_{Abst}, as required. For the queue invariant, the differences are that we no longer take the predicate Ψ, and we assert $G.\gamma_{\text{Abst}} \Rightarrow_\bullet xs_v$ instead of $\text{All}(xs_v, \Psi)$. That is, our queue invariant for the Hocap-style specification now looks as follows.

Definition 5.4.1 (Two-Lock M&S Queue Hocap Invariant)**.**

$$
\begin{array}{ll}
\mathrm{I_{TLH}}(\ell_{\mathrm{head}}, \ell_{\mathrm{tail}}, G) \triangleq & \\
\exists xs_v.\, G.\gamma_{\mathrm{Abst}} \Rightarrow_\bullet xs_v * & \text{(abstract state)} \\
\exists xs, xs_{\mathrm{queue}}, xs_{\mathrm{old}}, x_{\mathrm{head}}, x_{\mathrm{tail}}. & \text{(concrete state)} \\
xs = xs_{\mathrm{queue}} +\!\!+ [x_{\mathrm{head}}] +\!\!+ xs_{\mathrm{old}} * & \\
\mathrm{isLL}(xs) * & \\
\mathrm{projVal}(xs_{\mathrm{queue}}) = \mathrm{wrapSome}(xs_v) * & \\
(& \\
\quad \ell_{\mathrm{head}} \mapsto \mathrm{in}(x_{\mathrm{head}}) * \ell_{\mathrm{tail}} \mapsto \mathrm{in}(x_{\mathrm{tail}}) * \mathrm{isLast}(x_{\mathrm{tail}}, xs) * & \textbf{(Static)} \\
\quad \mathrm{TokNE}\ G * \mathrm{TokND}\ G * \mathrm{TokUpdated}\ G & \\
\quad \vee & \\
\quad \ell_{\mathrm{head}} \mapsto \mathrm{in}(x_{\mathrm{head}}) * \ell_{\mathrm{tail}} \mapsto^{\frac{1}{2}} \mathrm{in}(x_{\mathrm{tail}}) * & \textbf{(Enqueue)} \\
\quad (\mathrm{isLast}(x_{\mathrm{tail}}, xs) * \mathrm{TokBefore}\ G \vee \mathrm{isSndLast}(x_{\mathrm{tail}}, xs) * \mathrm{TokAfter}\ G) * & \\
\quad \mathrm{TokE}\ G * \mathrm{TokND}\ G & \\
\quad \vee & \\
\quad \ell_{\mathrm{head}} \mapsto^{\frac{1}{2}} \mathrm{in}(x_{\mathrm{head}}) * \ell_{\mathrm{tail}} \mapsto \mathrm{in}(x_{\mathrm{tail}}) * \mathrm{isLast}(x_{\mathrm{tail}}, xs) * & \textbf{(Dequeue)} \\
\quad \mathrm{TokNE}\ G * \mathrm{TokD}\ G * \mathrm{TokUpdated}\ G & \\
\quad \vee & \\
\quad \ell_{\mathrm{head}} \mapsto^{\frac{1}{2}} \mathrm{in}(x_{\mathrm{head}}) * \ell_{\mathrm{tail}} \mapsto^{\frac{1}{2}} \mathrm{in}(x_{\mathrm{tail}}) * & \textbf{(Both)} \\
\quad (\mathrm{isLast}(x_{\mathrm{tail}}, xs) * \mathrm{TokBefore}\ G \vee \mathrm{isSndLast}(x_{\mathrm{tail}}, xs) * \mathrm{TokAfter}\ G) * & \\
\quad \mathrm{TokE}\ G * \mathrm{TokD}\ G & \\
) &
\end{array}
$$

For the queue predicate, the only real difference is that we no longer take the predicate Ψ. That is, our queue predicate is the following.

Definition 5.4.2 (Two-Lock M&S Queue - isQueue Predicate (Hocap))**.**

$$
\begin{aligned}
\mathrm{isQueue_C}(v_q, G) \triangleq & \exists \ell_{\mathrm{queue}}, \ell_{\mathrm{head}}, \ell_{\mathrm{tail}} \in \mathit{Loc}.\, \exists h_{\mathrm{lock}}, t_{\mathrm{lock}} \in \mathit{Val}. \\
& v_q = \ell_{\mathrm{queue}} * \ell_{\mathrm{queue}} \mapsto^{\square} ((\ell_{\mathrm{head}}, \ell_{\mathrm{tail}}), (h_{\mathrm{lock}}, t_{\mathrm{lock}})) * \\
& \boxed{\mathrm{I_{TLH}}(\ell_{\mathrm{head}}, \ell_{\mathrm{tail}}, G)}^{\mathcal{N}.queue} * \\
& \mathrm{isLock}(G.\gamma_{\mathrm{Hlock}}, h_{\mathrm{lock}}, \mathrm{TokD}\ G) * \\
& \mathrm{isLock}(G.\gamma_{\mathrm{Tlock}}, t_{\mathrm{lock}}, \mathrm{TokE}\ G).
\end{aligned}
$$

5.4.3 Proof Sketch

The Hocap-style proofs are largely similar their concurrent counterparts. However, instead of having to handle the Ψ predicate, we must now 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 empty abstract state, and for enqueue and dequeue, the only real changes happen at the linearisation points. We sketch these challenges below.

Initialise

Lemma 12 (Two-Lock M&S Queue Hocap Specification - Initialise)**.**

$$\{\mathsf{True}\}\ \mathrm{initialize}\ ()\ \{v_q.\exists G.\ \mathrm{isQueue}(v_q, G) * G.\gamma_{\mathrm{Abst}} \Rightarrow_\circ []\}$$

As discussed, we must obtain $G.\gamma_{\mathrm{Abst}} \Rightarrow_\bullet [] * G.\gamma_{\mathrm{Abst}} \Rightarrow_\circ []$, for some ghost name $G.\gamma_{\mathrm{Abst}}$. We achieve this by applying lemma 1 instantiated with the empty state: $[]$. We use $G.\gamma_{\mathrm{Abst}} \Rightarrow_\bullet []$ to establish the queue invariant, and $G.\gamma_{\mathrm{Abst}} \Rightarrow_\circ []$ to prove the post-condition.

Enqueue

Lemma 13 (Two-Lock M&S Queue Hocap Specification - Enqueue)**.**

$$\forall v_q, v, G, P, Q.\ \left(\forall xs_v.\, G.\gamma_{\text{Abst}} \Mapsto_\bullet xs_v * P \Rrightarrow_{\mathcal{E}\setminus\mathcal{N}.i\uparrow} \triangleright G.\gamma_{\text{Abst}} \Mapsto_\bullet (v :: xs_v) * Q\right) \mathrel{-\!\!*} \{\text{isQueue}(v_q, G) * P\}\ \text{enqueue}\ \ v_q\ v\ \{w.Q\}$$

We start by assuming the view-shift which allows us to update P to Q and $G.\gamma_{\text{Abst}} \Mapsto_\bullet xs_v$ to $G.\gamma_{\text{Abst}} \Mapsto_\bullet (v :: xs_v)$, for any xs_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 simply frame away the newly added authoritative fragment, and continue as we did before. The second time we open the invariant is on line 10, around the expression that adds the newly created node, x_{new}, to the linked list. When opening the invariant, we now get $G.\gamma_{\text{Abst}} \Mapsto_\bullet xs_v$. And 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 $G.\gamma_{\text{Abst}} \Mapsto_\bullet (v :: xs_v)$. However, since we have $G.\gamma_{\text{Abst}} \Mapsto_\bullet xs_v$ and P (from the pre-condition), we can apply the view-shift 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 now additionally prove the post-condition Q, but this is no issue as we obtained that from the view-shift.

Dequeue

Lemma 14 (Two-Lock M&S Queue Hocap Specification - Dequeue)**.**

$$\forall v_q, G, P, Q.\ \left(\forall xs_v.\, G.\gamma_{\text{Abst}} \Mapsto_\bullet xs_v * P \Rrightarrow_{\mathcal{E}\setminus\mathcal{N}.i\uparrow} \triangleright \left(\begin{array}{l}(xs_v = [] * G.\gamma_{\text{Abst}} \Mapsto_\bullet xs_v * Q(\text{None})) \\ \vee \left(\begin{array}{l}\exists v, xs'_v.\, xs_v = xs'_v \mathbin{+\!\!+} [v] * \\ G.\gamma_{\text{Abst}} \Mapsto_\bullet xs'_v * Q(\text{Some}\, v)\end{array}\right)\end{array}\right)\right) \mathrel{-\!\!*} \{\text{isQueue}(v_q, G) * P\}\ \text{dequeue}\ \ v_q\ \{w.Q(w)\}$$

We assume the view-shift and proceed as in the concurrent proof until we get to the second time we open the invariant. This invariant opening is on line 17 around the expression that reads the head node's out pointer. It is here that we figure out whether or not the queue is empty by doing case analysis on xs_{queue}. We open the invariant and do the same case analysis here.

Case xs_{queue} **is empty**: In the case that the queue is empty, the abstract state of the queue will not change. We thus apply the view-shift (we have P from the precondition and $G.\gamma_{\text{Abst}} \Mapsto_\bullet xs_v$ from the invariant), which gives us the consequent of the view-shift. 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. $xs_v = [] * G.\gamma_{\text{Abst}} \Mapsto_\bullet xs_v * Q(\text{None})$. We now proceed as before, this time giving up $G.\gamma_{\text{Abst}} \Mapsto_\bullet xs_v$ to close the invariant. After stepping through the code, we are left with proving the postcondition: $Q(\text{None})$, which we got from the view-shift.

Case xs_{queue} **is not empty**: If the queue is not empty, then we do not apply the view-shift (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 on line 23, around the expression that swings ℓ_{head} to $x_{\text{head\_next}}$. It is this store operation that updates the abstract state of the queue, so it is within this invariant opening that we apply the view-shift (again, we have P from the pre-condition and $G.\gamma_{\text{Abst}} \Mapsto_\bullet xs_v$ from the invariant). As we did previously, we deduce that the current 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 view-shift gives us $\exists v, xs'_v.\, xs_v = xs'_v \mathbin{+\!\!+} [v] * G.\gamma_{\text{Abst}} \Mapsto_\bullet xs'_v * Q(\text{Some}\, v)$. As before, we conclude that $\text{Some}\, v$ is the return value (through the reflection between xs_{queue} and xs_v), and proceed to close the invariant, this time giving up $G.\gamma_{\text{Abst}} \Mapsto_\bullet xs'_v$. Stepping through the code, we end up having to prove the post-condition $Q(\text{Some}\, v)$, which we got from the view-shift.

Chapter 6

The Lock-Free Michael-Scott Queue

6.1 Introduction

In this chapter we will study the non-blocking version of the M&S Queue, the Lock-Free M&S Queue. As with the two-lock version, the original implementation can be found in Michael and Scott [1996]. 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.

6.2 Implementation

The implementation shares many commonalities with the two-lock variant, most importantly, the underlying queue is still a linked list. The major differences are how we manipulate the linked list and the head and tail pointers.

6.2.1 Initialise

As the implementation doesn't use locks, the queue data structure is now just a pointer to a pair consisting of the head and tail pointers.

6.2.2 Enqueue

Enqueueing a node consists of the same two steps as before: add the newly created node to the linked list, and swing the tail pointer. In this way, figures 4.2a, 4.2b, and 4.2c still accurately reflect the state changes that the queue goes through during an enqueue. However, one big difference is that swinging the tail pointer to the newly inserted node is not necessarily done by the thread that enqueued it. We discuss this below.
Since other threads can work on the linked list at the same time, we must do some extra checks when enqueueing a new node.
Firstly, we must ensure that the tail pointer is actually pointing to the last node. We ensure this on line 11. If it isn't, that means that another enqueueing thread has added a new node to the linked list, but hasn't yet swung the tail pointer. So instead of waiting for it, we *help* it by trying to swing the tail pointer for it. Afterwards, we try to enqueue our node again.
Secondly, before we can add the node, we must ensure that no thread has performed an enqueue while we have been working, so that we don't overwrite another threads enqueue. This is ensured with the CAS instruction on line 12 – it adds the new node to the linked list only if our tail node is still the last, and hence points to None. After adding the node, we attempt to swing the tail pointer to it. This may fail, but that simply means that another thread has already swung it for us. Finally, we have an additional *consistency check* on line 10. We discuss the purpose of this extra check in section 7.6.

6.2.3 Dequeue

Fundamentally, a dequeue operation still consists of swinging the head pointer to the next node in the linked list. The original authors decided that the tail pointer shouldn't lag behind the head pointer, hence dequeue also accesses the tail node and ensures it won't lag behind as result of the dequeue. On line 26, we check whether the head and tail nodes are the same. If this is the case, then the queue is either empty or the tail pointer is lagging behind. We distinguish between these two cases on line 27. If the tail node is indeed lagging behind, then some thread has enqueued a node, but not yet swung the tail pointer, so we help it out by swinging the pointer.
If the head and tail nodes are not the same, then it must be safe to dequeue, which we attempt in the else block on line 31.

6.2.4 Prophecies

Notice that the load on line 24 is a possible linearisation point – if the load resolves to None (i.e. the queue is empty) and the consistency check on the next line passes, the dequeue will conclude the queue is empty and return None. That is, the point where we "read" the state of the queue is the load on line 24, but we only conclude that the queue is empty if the consistency check on the next line succeeds. So when we are at the load, we need to know whether or not the consistency check passes; if it does, we should apply the view-shift, and if it doesn't, we shouldn't apply it. This is somewhat of a conundrum, as this requires us to reason about a future computation.

To solve this issue, we use a *prophecy variable*. Prophecies are a part of Iris and allow us to reason about the result of future expressions. They are only used in the logic and do not alter the semantics of the code. On line 23 in dequeue, we create one such prophecy variable, which is then "resolved" on line 25. The prophecy variable allows us to reason about the result of the expression inside the Resolve statement. In other words, we can reason about consistency check already at the load on line 24, allowing us to make the correct choice. We will see exactly how this works when we prove the specification for the Lock-Free M&S Queuein chapter 7.

```
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_ =
    let head = !(fst(! Q)) in
    let tail = !(snd(! Q)) in
    let p = newproph in
    let next = !(snd(! head)) in
    if head = Resolve(!(fst(! Q)), p, ()) then
      if head = tail then
        if next = None then
          None
        else
          CAS(snd(! Q)) tail next; loop ()
      else
        let value = fst(! next) in
        if CAS (fst(! Q)) head next then
          value
        else loop ()
    else loop ()
  )()
```

Figure 6.1: Implementation of Lock-Free M&S Queue in HeapLang.

Chapter 7

Proving Hocap-style Specification for the Lock-Free Michael-Scott Queue

7.1 Introduction

In this chapter we prove that the Lock-Free M&S Queue satisfies the Hocap-style specification given in section 3.4. Section 7.2 introduces the notion of *Reachability* which we will need in the proofs. In section 7.3 we discuss the specification we will be proving, and in sections 7.4 and 7.5 we give the proofs. Finally, in section 7.6, we discuss a simplification to the Lock-Free M&S Queue algorithm and its implications.

7.2 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 26, 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 32, 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 Vindum and Birkedal [2021]. The presentation here borrows the same ideas, but 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.

7.2.1 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 concrete reachability as an inductive predicate.

Definition 7.2.1 (Concrete Reachability)**.** Given two nodes, x_n and x_m, we say that x_n concretely reaches x_m, if they satisfy the following inductive predicate.

$$x_n \rightsquigarrow x_m \triangleq \text{in}(x_n) \mapsto^{\square} (\text{val}(x_n), \text{out}(x_n)) * (x_n = x_m \vee \exists x_p.\ \text{out}(x_n) \mapsto^{\square} \text{in}(x_p) * x_p \rightsquigarrow x_m)$$

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. This in turn makes concrete reachability a persistent predicate.

We proceed to prove some useful lemmas about concrete reachability.

Lemma 15 (Reach Reflexive)**.** $x_n \rightsquigarrow x_n \mathrel{-\!\!*\!\!*} \text{in}(x_n) \mapsto^{\square} (\text{val}(x_n), \text{out}(x_n))$

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

Lemma 16 (Reach Transitive)**.** $x_n \rightsquigarrow x_m \mathrel{-\!\!*} x_m \rightsquigarrow x_o \mathrel{-\!\!*} x_n \rightsquigarrow x_o$

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 $x_m \rightsquigarrow x_o \mathrel{-\!\!*} x_p \rightsquigarrow x_o$. 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 17 (Reach From is Node)**.** $x_n \rightsquigarrow x_m \mathrel{-\!\!*} \text{in}(x_n) \mapsto^{\square} (\text{val}(x_n), \text{out}(x_n))$

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

Lemma 18 (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 17 above. In the inductive case, we assume that x_n points to some x_p, which reaches x_m. Our induction hypothesis is $\text{in}(x_m) \mapsto^{\square} (\text{val}(x_m), \text{out}(x_m))$, which is also our proof obligation, so we are done. □

Lemma 19 (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 $\text{out}(x_n) \mapsto \text{None}$, we must prove that $x_n = x_m$ and $\text{out}(x_n) \mapsto \text{None}$. 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 $\text{out}(x_n) \mapsto^{\square} \text{in}(x_p)$. But by our initial assumption, $\text{out}(x_n) \mapsto \text{None}$. Since $\text{in}(x_p)$ is a location, then this is clearly a contradiction. □

7.2.2 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 $x \rightsquigarrow z$.

We refer to section 8.3 for the definition of the resource algebra we use to formally define abstract points-to and abstract reachability. For our purposes, we only need to know the following four lemmas to work with the predicates in the proofs. These lemmas are also proved in section 8.3.

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 20 (Abs Reach Alloc)**.** For all nodes x, we have

$$x \rightsquigarrow x \Rrightarrow (\exists \gamma.\, \gamma \rightarrowtail x * x \dashrightarrow \gamma)$$

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 21 (Abs Reach Concr)**.** For nodes x_n and x_m and ghost names γ_m, we have

$$x_n \dashrightarrow \gamma_m * \gamma_m \rightarrowtail x_m \mathrel{-\!\!*} x_n \rightsquigarrow x_m * \gamma_m \rightarrowtail x_m$$

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 22 (Abs Reach Abs)**.** For nodes x_n and x_m and ghost names γ_m, we have

$$x_n \rightsquigarrow x_m * \gamma_m \rightarrowtail x_m \Rrightarrow x_n \dashrightarrow \gamma_m * \gamma_m \rightarrowtail x_m$$

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 23 (Abs Reach Advance)**.** For nodes x_m and x_o and ghost names γ_m, we have

$$\gamma_m \rightarrowtail x_m * x_m \rightsquigarrow x_o \Rrightarrow \gamma_m \rightarrowtail x_o * x_o \dashrightarrow \gamma_m$$

7.3 Specifications for Lock-Free M&S Queue

As we showed in section 3.5, the sequential and concurrent specifications can be derived from the Hocap-style specification *without* referring to the actual implementation. Thus, in this chapter we focus only on proving the Hocap-style specification, and rely on the derivations to prove the concurrent and sequential specifications.

We prove the Hocap-style specification for the initialize, enqueue, and dequeue functions introduced in chapter 6. We make the collection of ghost names, *Qgnames*, be the set of four-tuples. Firstly, it contains the mandatory γ_{Abst} whose purpose is the same as before; to keep track of the abstract state of the queue. Additionally, it contains γ_{Head}, γ_{Tail}, and γ_{Last}, which will abstractly point to the head, tail, and last node, respectively.

7.4 Hocap-style Queue Predicate

Since the queue predicate is required to be persistent and multiple threads need to access shared resources of the queue, we will be needing an invariant. 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 does not 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 7.4.1 (Lock-Free M&S Queue Invariant)**.**

$$
\begin{aligned}
&\mathrm{I}_{\mathrm{LFH}}(\ell_{\text{head}}, \ell_{\text{tail}}, G) \triangleq \\
&\exists xs_v.\, G.\gamma_{\text{Abst}} \Mapsto_{\bullet} xs_v * && \text{(abstract state)} \\
&\exists xs, xs_{\text{queue}}, x_{\text{head}}, x_{\text{tail}} x_{\text{last}}. && \text{(concrete state)} \\
&xs = xs_{\text{queue}} \mathbin{+\!\!+} [x_{\text{head}}] * \\
&\mathrm{isLL}(xs) * \\
&\mathrm{isLast}(x_{\text{last}}, xs) \\
&\mathrm{projVal}(xs_{\text{queue}}) = \mathrm{wrapSome}(xs_v) * \\
&\ell_{\text{head}} \mapsto \mathrm{in}(x_{\text{head}}) * \\
&\ell_{\text{tail}} \mapsto \mathrm{in}(x_{\text{tail}}) * \\
&G.\gamma_{\text{Head}} \rightarrowtail x_{\text{head}} * x_{\text{head}} \dashrightarrow G.\gamma_{\text{Tail}} * \\
&G.\gamma_{\text{Tail}} \rightarrowtail x_{\text{tail}} * x_{\text{tail}} \dashrightarrow G.\gamma_{\text{Last}} * \\
&G.\gamma_{\text{Last}} \rightarrowtail x_{\text{last}}
\end{aligned}
$$

The isQueue 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 7.4.2 (Lock-Free M&S Queue - isQueue Predicate)**.**

$$
\begin{aligned}
\mathrm{isQueue}(v_q, G) \triangleq &\exists \ell_{\text{queue}}, \ell_{\text{head}}, \ell_{\text{tail}} \in \mathit{Loc}. \\
&v_q = \ell_{\text{queue}} * \ell_{\text{queue}} \mapsto^{\square} (\ell_{\text{head}}, \ell_{\text{tail}}) * \\
&\boxed{\mathrm{I}_{\mathrm{LFH}}(\ell_{\text{head}}, \ell_{\text{tail}}, G)}^{\mathcal{N}.queue}
\end{aligned}
$$

7.5 Proof Outline

We instantiate the specification with our definition of isQueue (7.4.2). By the definition of isQueue we easily show that isQueue 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 13 and 15, and for dequeue, line 30). These all behave similarly, so we additionally prove a specification for swinging the tail.

Initialise

Lemma 24 (Lock-Free M&S Queue Specification - Initialise)**.**

$$\{\mathsf{True}\}\ \mathrm{initialize}\ ()\ \{v_q.\exists G.\ \mathrm{isQueue}(v_q, G) * G.\gamma_{\text{Abst}} \Mapsto_{\circ} []\}$$

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 $\ell_{\text{head}} \mapsto \mathrm{in}(x_1)$ and $\ell_{\text{tail}} \mapsto \mathrm{in}(x_1)$, and finally $\ell_{\text{queue}} \mapsto (\ell_{\text{head}}, \ell_{\text{tail}})$.

As we did for the two-lock version, we apply lemma 1 to allocate an empty abstract queue, giving us some ghost name γ_{Abst} that we put into G, and the resources $G.\gamma_{\text{Abst}} \Mapsto_\bullet [] * G.\gamma_{\text{Abst}} \Mapsto_\circ []$. To allocate the invariant, we must additionally obtain abstract reach propositions. Since x_1 is a node, we may use lemma 15 to conclude $x_1 \rightsquigarrow x_1$. We can now use lemma 20 three times, giving us ghost names γ_{Head}, γ_{Tail}, γ_{Last} which we again put into G, and the resources

$$G.\gamma_{\text{Head}} \rightarrowtail x_1 * G.\gamma_{\text{Tail}} \rightarrowtail x_1 * x_1 \dashrightarrow G.\gamma_{\text{Tail}} * G.\gamma_{\text{Last}} \rightarrowtail x_1 * x_1 \dashrightarrow G.\gamma_{\text{Last}}$$

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 25 (Swing Tail)**.** $\forall \ell_{\text{head}}, \ell_{\text{tail}}, x_{\text{tail}}, x_{\text{newtail}}, G.$
$\left\{\boxed{\text{I}_{\text{LFH}}(\ell_{\text{head}}, \ell_{\text{tail}}, G)}^{\mathcal{N}.queue} * x_{\text{tail}} \rightsquigarrow x_{\text{newtail}} * x_{\text{newtail}} \dashrightarrow G.\gamma_{\text{Last}}\right\}$
CAS ℓ_{tail} $\text{in}(x_{\text{tail}})$ $\text{in}(x_{\text{newtail}})$
$\{w.w = \text{true} \vee w = \text{false}\}$

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 $\ell_{\text{tail}} \mapsto \text{in}(x'_{\text{tail}})$. We consider both cases of the CAS:

Case CAS succeeds. It must then have been the case that $\text{in}(x'_{\text{tail}}) = \text{in}(x_{\text{tail}})$. Since we have $x_{\text{tail}} \rightsquigarrow x_{\text{newtail}}$, we know that x_{tail} is a node (lemma 17). From the invariant, we additionally got that $G.\gamma_{\text{Tail}} \rightarrowtail x'_{\text{tail}}$ and $x_{\text{head}} \dashrightarrow G.\gamma_{\text{Tail}}$, which by lemma 21 means that $x_{\text{head}} \rightsquigarrow x'_{\text{tail}}$. We can thus also conclude that x'_{tail} is a node (lemma 18). So since both x_{tail} and x'_{tail} are nodes, and $\text{in}(x'_{\text{tail}}) = \text{in}(x_{\text{tail}})$, lemma 33 tells us that $x_{\text{tail}} = x'_{\text{tail}}$. In other words, we have that $G.\gamma_{\text{Tail}} \rightarrowtail x_{\text{tail}}$. Since the CAS succeeded, we now have that $\ell_{\text{tail}} \mapsto \text{in}(x_{\text{newtail}})$. Since the invariant demands that $G.\gamma_{\text{Tail}}$ and ℓ_{tail} agree on the node they point to, we must update $G.\gamma_{\text{Tail}} \rightarrowtail x_{\text{tail}}$ to $G.\gamma_{\text{Tail}} \rightarrowtail x_{\text{newtail}}$. We can do this using lemma 23 as we assumed $x_{\text{tail}} \rightsquigarrow x_{\text{newtail}}$. 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 26 (Lock-Free M&S Queue Specification - Enqueue)**.**

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

Proof. We assume the view-shift and proceed to prove the Hoare triple. By definition of isQueue, we know that v_q is a location ℓ_{queue} and there are locations ℓ_{head} and ℓ_{tail} so that

$$\ell_{\text{queue}} \mapsto^{\square} (\ell_{\text{head}}, \ell_{\text{tail}}) * \boxed{\text{I}_{\text{LFH}}(\ell_{\text{head}}, \ell_{\text{tail}}, G)}^{\mathcal{N}.queue} \tag{7.1}$$

We first step through line 6 which creates a new node x_{new}, so that

$$\text{in}(x_{\text{new}}) \mapsto^{\square} (\text{Some}\, v, \text{out}(x_{\text{new}})) \tag{7.2}$$
$$\text{out}(x_{\text{new}}) \mapsto \text{None} \tag{7.3}$$

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 8 first dereferences to the tail pointer ℓ_{tail} using the resources in 7.1. 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:

$$x_{\text{tail}} \dashrightarrow G.\gamma_{\text{Last}} \tag{7.4}$$
$$\text{in}(x_{\text{tail}}) \mapsto^{\square} (\text{val}(x_{\text{tail}}), \text{out}(x_{\text{tail}})) \tag{7.5}$$

The first part is directly from the invariant, and the second we may derive using lemmas 21 and 18. We perform the load, and close the invariant.
The next line (line 9) finds out what x_{tail} points to. Using 7.5 we step to $!(\text{out}(x_{\text{tail}}))$. 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 $G.\gamma_{\text{Last}} \rightarrowtail x_{\text{last}}$. From this, 7.4, and lemma 21 we conclude $x_{\text{tail}} \rightsquigarrow x_{\text{last}}$. 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 $x_{\text{tail\_next}}$ which can reach x_{last} (meaning that x_{tail} is lagging behind).

Case $x_{\text{tail}} = x_{\text{last}}$. Since we had $\text{isLast}(x_{\text{last}}, xs)$, 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 10. As before, the points-to predicate for ℓ_{tail} is in the invariant, so we open it. We get $\ell_{\text{tail}} \mapsto \text{in}(x'_{\text{tail}})$, 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 16, 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 11. Here we check whether or not *next* is None. We already know this is the case, so we proceed to line 12. 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 21 to conclude $x_{\text{tail}} \rightsquigarrow x'_{\text{last}}$, where x'_{last} is the current last node of the linked list (according to the invariant). As before, we perform case analysis on $x_{\text{tail}} \rightsquigarrow x'_{\text{last}}$.

Case $x_{\text{tail}} = x'_{\text{last}}$. We now know that x_{tail} is still the last node in the linked list, hence $\text{out}(x_{\text{tail}}) \mapsto$ 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 view-shift. We instantiate the view-shift with the abstract state of the queue xs_v from the invariant opening, and supply $G.\gamma_{\text{Abst}} \Rightarrow_{\bullet} xs_v$ from the invariant and the P from the pre-condition. We hence get $G.\gamma_{\text{Abst}} \Rightarrow_{\bullet} (v :: xs_v)$ and Q.
When closing the invariant, we use $(v :: xs_v)$ for the abstract state, $(x_{\text{new}} :: xs)$ for the concrete state, $x_{\text{new}} :: xs_{\text{queue}}$ 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 $G.\gamma_{\text{Abst}} \Rightarrow_{\bullet} (v :: xs_v)$ that we got from the view-shift, and the points-to predicate 7.3 (used to assert $\text{isLL}(x_{\text{new}} :: xs)$). The only thing left to prove is $G.\gamma_{\text{Last}} \rightarrowtail x_{\text{new}}$. From the invariant opening, we have $G.\gamma_{\text{Last}} \rightarrowtail x_{\text{tail}}$. Since $x_{\text{tail}} \rightsquigarrow x_{\text{new}}$, we may apply lemma 23 to update the abstract points-to resource to $G.\gamma_{\text{Last}} \rightarrowtail x_{\text{new}}$ and additionally obtain $x_{\text{new}} \dashrightarrow G.\gamma_{\text{Last}}$. With this, we can close the invariant, and step to line 13. This line attempts to swing the tail, so we apply our swing-tail lemma (lemma 25) by supplying our invariant, $x_{\text{tail}} \rightsquigarrow x_{\text{new}}$, and $x_{\text{new}} \dashrightarrow G.\gamma_{\text{Last}}$. 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 $\text{out}(x_{\text{tail}}) \mapsto^{\square} \text{in}(x_{\text{tail\_next}}) * x_{\text{tail\_next}} \rightsquigarrow x_{\text{last}}$. Since x_{tail} doesn't point to None, the CAS will fail. We close the invariant and step to line 14. We finish by applying the induction hypothesis.

Case $\text{out}(x_{\text{tail}}) \mapsto^{\square} \text{in}(x_{\text{tail\_next}}) * x_{\text{tail\_next}} \rightsquigarrow x_{\text{last}}$. Using this we perform the load, which sets *next* to $\text{in}(x_{\text{tail\_next}})$. Before closing the invariant, we apply lemma 22 with $x_{\text{tail\_next}} \rightsquigarrow x_{\text{last}}$ and

$G.\gamma_{\text{Last}} \rightarrowtail x_{\text{last}}$ to obtain $x_{\text{tail\_next}} \dashrightarrow G.\gamma_{\text{Last}}$. 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 11. This time, the check will fail as *next* is $\text{in}(x_{\text{tail\_next}})$ which is not None. Hence we step to line 15 which attempts to swing the tail pointer. We here apply lemma 25 which we can do as we own the invariant, $x_{\text{tail}} \rightsquigarrow x_{\text{tail\_next}}$, and $x_{\text{tail\_next}} \dashrightarrow G.\gamma_{\text{Last}}$. We step through to the recursive call and finish by applying the induction hypothesis.

□

Dequeue

Lemma 27 (Lock-Free M&S Queue Specification - Dequeue)**.**

$$\forall v_q, G, P, Q.$$
$$\left(\forall xs_v.\, G.\gamma_{\text{Abst}} \Mapsto_\bullet xs_v * P \Rrightarrow_{\mathcal{E}\setminus\mathcal{N}.i\uparrow} \triangleright \left(\begin{array}{l}(xs_v = [] * G.\gamma_{\text{Abst}} \Mapsto_\bullet xs_v * Q(\text{None})) \\ \vee \left(\begin{array}{l}\exists v, xs'_v.\, xs_v = xs'_v \mathbin{+\!\!+} [v] * \\ G.\gamma_{\text{Abst}} \Mapsto_\bullet xs'_v * Q(\text{Some}\, v)\end{array}\right)\end{array}\right)\right) \mathrel{-\!\!*}$$
$$\{\text{isQueue}(v_q, G) * P\}\ \text{dequeue}\ v_q\ \{w.Q(w)\}$$

Proof. We assume the view-shift and must prove the Hoare triple. As before, we know from isQueue that the queue, v_q, is a location, ℓ_{queue}, and there are locations ℓ_{head} and ℓ_{tail} so that

$$\ell_{\text{queue}} \mapsto^{\square} (\ell_{\text{head}}, \ell_{\text{tail}}) * \boxed{\text{I}_{\text{LFH}}(\ell_{\text{head}}, \ell_{\text{tail}}, G)}^{\mathcal{N}.queue} \tag{7.6}$$

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 21. 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 $\text{in}(x_{\text{head}})$. We also derive the following information

$$\text{in}(x_{\text{head}}) \mapsto^{\square} (\text{val}(x_{\text{head}}), \text{out}(x_{\text{head}})) \tag{7.7}$$
$$x_{\text{head}} \dashrightarrow G.\gamma_{\text{Head}} \tag{7.8}$$
$$x_{\text{head}} \dashrightarrow G.\gamma_{\text{Tail}} \tag{7.9}$$
$$x_{\text{head}} \dashrightarrow G.\gamma_{\text{Last}} \tag{7.10}$$

From the abstract points-to predicates from the invariant and lemma 21 we get that $x_{\text{head}} \rightsquigarrow x_{\text{tail}}$, so by lemma 17, we know that x_{head} is a node. This shows 7.7. By reflexivity of reach (lemma 15) we additionally know that $x_{\text{head}} \rightsquigarrow x_{\text{head}}$. Lemma 22 then gives us 7.8 and 7.9.
Lastly, we use lemma 21 to deduce that $x_{\text{tail}} \rightsquigarrow x_{\text{last}}$. By transitivity of reach (lemma 16) we have that $x_{\text{head}} \rightsquigarrow x_{\text{last}}$, which we use with lemma 22 to get 7.10.
We now close the invariant and step to line 22 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 7.9 and lemmas 21, 18, and 22 we get the following

$$\text{in}(x_{\text{tail}}) \mapsto^{\square} (\text{val}(x_{\text{tail}}), \text{out}(x_{\text{tail}})) \tag{7.11}$$
$$x_{\text{head}} \rightsquigarrow x_{\text{tail}} \tag{7.12}$$
$$x_{\text{tail}} \dashrightarrow G.\gamma_{\text{Tail}} \tag{7.13}$$
$$\tag{7.14}$$

We close the invariant and step to line 23. This line creates our *prophecy variable*, p, which will be resolved on line 25. This allows us to reason about the result of the consistency check: we will later show that the expression associated with p (i.e. $!(\mathsf{fst}(!\,Q))$) evaluates to some value v_p, but we can already now case on whether v_p will be equal to $\text{in}(x_{\text{head}})$ – the left hand side of the equality check on line 25.

Case $\text{in}(x_{\text{head}}) = v_p$. We continue to line 24, 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

$$G.\gamma_{\text{Last}} \rightarrowtail x_{\text{last}} \tag{7.15}$$

Combining this with 7.10 and lemma 21 we conclude $x_{\text{head}} \rightsquigarrow x_{\text{last}}$. This gives us two cases to consider: either $x_{\text{head}} = x_{\text{last}}$ or x_{head} points to some $x_{\text{head\_next}}$, which reaches x_{last}.

Case $x_{\text{head}} = x_{\text{last}}$. This corresponds to the queue being empty, which we derive below.
As x_{last} is the last node, we have that $\text{out}(x_{\text{head}}) \mapsto \text{None}$, hence the load resolves to None. Using the abstract points-to predicates from the invariant together with 7.8, 7.9, and lemma 21 we get $x_{\text{head}} \rightsquigarrow x'_{\text{head}}$ and $x_{\text{head}} \rightsquigarrow x'_{\text{tail}}$. We can now apply lemma 19 three times to conclude $x_{\text{head}} = x'_{\text{head}} = x'_{\text{tail}} = x_{\text{tail}}$. 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 $x_{\text{head}} = x_{\text{tail}}$, we know 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 $G.\gamma_{\text{Abst}} \Rrightarrow_\bullet []$. We thus instantiate the view-shift 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 $Q(\text{None})$ and $G.\gamma_{\text{Abst}} \Rrightarrow_\bullet []$. As we haven't changed any resources, we can close the invariant again.
We reach the consistency check on line 25. By 7.6, we know that $!(\mathsf{fst}(!\,Q))$ steps to $!(\ell_{\text{head}})$, but to resolve the prophecy, we must first show what $!(\ell_{\text{head}})$ evaluates to. This resource is inside the invariant so we open it. We get that $\ell_{\text{head}} \mapsto \text{in}(x''_{\text{head}})$ for some node x''_{head}. We close the invariant and resolve the prophecy: $!(\mathsf{fst}(!\,Q))$ evaluated to $\text{in}(x''_{\text{head}})$. In other words, $v_p = \text{in}(x''_{\text{head}})$, and therefore $\text{in}(x_{\text{head}}) = \text{in}(x''_{\text{head}})$. Since the remaining if statement on line 25 compares $\text{in}(x_{\text{head}})$ to $\text{in}(x''_{\text{head}})$, we know that we will take the "then" branch, so we step to line 26. Since $x_{\text{head}} = x_{\text{tail}}$ and *next* was set to None, we step to line 28 which returns None. The post-condition thus requires us to prove $Q(\text{None})$, which we already have.

Case $\text{out}(x_{\text{head}}) \mapsto^{\square} \text{in}(x_{\text{head\_next}}) * x_{\text{head\_next}} \rightsquigarrow x_{\text{last}}$. This means that the queue is not empty, and there is an element to be dequeued: the value in $x_{\text{head\_next}}$. The load resolves to $\text{in}(x_{\text{head\_next}})$, and the program variable *next* is set to this. Using lemmas 17 and 22 with 7.15 we get

$$\text{in}(x_{\text{head\_next}}) \mapsto^{\square} (\text{val}(x_{\text{head\_next}}), \text{out}(x_{\text{head\_next}})) \tag{7.16}$$

$$x_{\text{head\_next}} \dashrightarrow G.\gamma_{\text{Last}} \tag{7.17}$$

We close the invariant and step to the consistency check on line 25. We handle this similarly to the previous case and conclude that the consistency check succeeds, taking us to line 26 in the "then" branch. 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 30, which attempts to swing the tail, and try dequeueing again. We handle the CAS with our swing-tail lemma (lemma 25), 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 31 which attempts to dequeue. We first read the value out of $x_{\text{head\_next}}$ on line 32. Next, we attempt to swing the head pointer on line 33. 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 $\ell_{\text{head}} \mapsto \text{in}(x'_{\text{head}})$. The success of the CAS depends on whether $\text{in}(x'_{\text{head}})$ equals $\text{in}(x_{\text{head}})$. 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 35 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 $x_{\text{head\_next}}$, the queue data structure got updated, so this is a linearisation point.
Using the abstract points-to and reachability propositions from the invariant together with lemmas 21 and 17, we may deduce that x'_{head} is a node. As we already know that x_{head}

is a node (from 7.7), lemma 33 tells us that the nodes are equal: $x_{\text{head}} = x'_{\text{head}}$. From the invariant (specifically isLL(xs)) we know that x'_{head} points to the first element of xs_{queue}. But since $x_{\text{head}} = x'_{\text{head}}$, then this element must be our $x_{\text{head\_next}}$. In other words, $xs_{\text{queue}} = xs'_{\text{queue}} +\!\!+ [x_{\text{head\_next}}]$, for some xs'_{queue}. This means that, when we apply the view-shift (giving up P and $G.\gamma_{\text{Abst}} \Rightarrow\!\!\!\bullet\ xs_v$ 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 v so that

$$xs_v = xs'_v +\!\!+ [v] \tag{7.18}$$
$$G.\gamma_{\text{Abst}} \Rightarrow\!\!\!\bullet\ xs'_v \tag{7.19}$$
$$Q(\text{Some}\, v) \tag{7.20}$$

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 $\text{Some}\, v = \text{val}(x_{\text{head\_next}})$.
To close the invariant, we must update some of our resources. Since we now have $\ell_{\text{head}} \mapsto \text{in}(x_{\text{head\_next}})$, we must pick $x_{\text{head\_next}}$ for the head node. But currently $G.\gamma_{\text{Head}} \rightarrowtail x_{\text{head}}$. So we use lemma 23 to advance the pointer, and we get $G.\gamma_{\text{Head}} \rightarrowtail x_{\text{head\_next}}$.
We are now required to show that $x_{\text{head\_next}} \dashrightarrow G.\gamma_{\text{Tail}}$. Since $x_{\text{head}} \rightsquigarrow x_{\text{tail}}$, and $x_{\text{head}} \neq x_{\text{tail}}$, then it must be the case that $x_{\text{head\_next}} \rightsquigarrow x_{\text{tail}}$. Lemma 21 with 7.13 now tells us that x_{tail} can reach the current tail node, x'_{tail}. By transitivity (lemma 16), we get $x_{\text{head\_next}} \rightsquigarrow x'_{\text{tail}}$, and hence by lemma 22, we get the desired $x_{\text{head\_next}} \dashrightarrow G.\gamma_{\text{Tail}}$. We now own all the resources required to close the invariant.
As the CAS succeed, we step to line 34 which simply returns $\text{val}(x_{\text{head\_next}})$. Thus, we must prove the post-condition $Q(\text{val}(x_{\text{head\_next}}))$. We can do this since we still own 7.20 and we deduced that $\text{Some}\, v = \text{val}(x_{\text{head\_next}})$.

Case $\text{in}(x_{\text{head}}) \neq v_p$. We step to line 24 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 $G.\gamma_{\text{Last}} \rightarrowtail x_{\text{last}}$. By combining this with 7.10 and lemma 21 we deduce that $x_{\text{head}} \rightsquigarrow x_{\text{last}}$ which means that either x_{head} *is* the last node, and hence $\text{out}(x_{\text{head}}) \mapsto \text{None}$, or there is some other node $x_{\text{head\_next}}$ and $\text{out}(x_{\text{head}}) \mapsto^{\square} x_{\text{head\_next}}$. 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 25.
By 7.6, we know that $!(\mathsf{fst}(!\,Q))$ steps to $!(\ell_{\text{head}})$, so to resolve the prophecy, we show what $!(\ell_{\text{head}})$ evaluates to. We open the invariant, which gives us that $\ell_{\text{head}} \mapsto \text{in}(x'_{\text{head}})$ for some node x'_{head}. We close the invariant again, and resolve the prophecy: $!(\mathsf{fst}(!\,Q))$ evaluated to $\text{in}(x'_{\text{head}})$. That is, $v_p = \text{in}(x'_{\text{head}})$, and therefore $\text{in}(x_{\text{head}}) \neq \text{in}(x'_{\text{head}})$. We hence take the "else" branch and step to line 36. This consists of a recursive call to the loop function and we are done by the induction hypothesis.

□

7.6 Discussion

It was shown in Vindum and Birkedal [2021] that a version of the Lock-Free M&S Queue with the consistency checks is contextually equivalent to a version without consistency checks. In the original presentation [Michael and Scott, 1996] the implementation language was assumed to not have a garbage collector. This meant that the ABA problem was an issue; if a node is freed by a dequeue, and afterwards a new node is allocated at the same location with the same value through an enqueue, it will look as though it is the exact same node. This can cause inconsistencies for threads that read the original node, went to sleep, and continued after the new allocation. To fix the issue, the authors added *modification counters* to their pointers, which means that the pointer to the newly allocated node will have a higher counter, and we can hence tell that it isn't the same node as the original. The essence of the consistency checks is to ensure that previously read nodes are the exact same nodes as from before, by ensuring that the counter is the same. We do this check *after* reading any "next" nodes to ensure that they are indeed the next of the node we originally read – the original node and its next node are consistent.

However, the language used in Vindum and Birkedal [2021] and this project, HeapLang, is a garbage-collected language. This means that we do not need to worry about freeing nodes, and the ABA problem doesn't occur. In other words, when we read the "next" of a node, we can be certain that it is the next of the node we originally read – no one could have freed it in a dequeue it and subsequently allocated a similar looking node in an enqueue. This then means that the consistency checks are no longer needed.

Indeed, in the Coq formalisations, we prove the Hocap-style specification for an implementation that doesn't have the consistency checks. As an additional benefit, removing the consistency check means that in dequeue, we know already when we read the "next" of the head node on line 24 whether or not the dequeue will conclude that the queue is empty, and hence whether or not the load is a linearisation point. The prophecy variable is thus not needed, which further simplifies the proof of the specification.

Chapter 8

On the Resource Algebras Used

8.1 Tokens

Tokens are defined using the exclusive resource algebra on the singleton set: Ex(). This resource algebra has only one valid element, which we here denote by T, and combining two elements will give the non-valid element $\bot$. The definition of tokens is then quite simple.

Definition 8.1.1 (Token)**.** $\text{Token}(\gamma) = \boxed{T}^{\gamma}$

We can always create new tokens, which will just give us ownership of T with a fresh ghost name, γ.

Lemma 7 (Token Alloc)**.** $\vdash \Rrightarrow \exists \gamma.\ \text{Token}(\gamma)$

Proof. This follows directly from rule Ghost-alloc and the fact that T is valid. □

Importantly, if we own a particular token, then we know that no one else can own the same token. Formally, we write this as:

Lemma 8 (Token Exclusive)**.** $\text{Token}(\gamma) * \text{Token}(\gamma) \vdash \mathsf{False}$

Proof. By Own-op we get ownership of $\boxed{T \cdot T}^{\gamma}$. By the exclusive RA, this is equivalent to $\boxed{\bot}^{\gamma}$. By rule Own-valid, we must have that $\bot \in \mathcal{V}$. This is a contradiction as bot is a non-valid element. □

8.2 Abstract State

We use the resource algebra $\textsc{Auth}((\textsc{Frac} \times \textsc{Ag}(\textit{List Val}))^{?})$ to define the abstract state predicate. *List Val* is the abstract state. It is wrapped in the agreement RA, Ag, which ensures that if one owns two elements, then they agree on the abstract state. The fractional RA, 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 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 Auth construction. Auth is the authoritative resource algebra. It gives us the authoritative and fragmental views and, together with the fractional RA, governs that they can only be updated in unison.
We note that core of the fractional RA is always undefined, which means that the core of $\textsc{Auth}((\textsc{Frac} \times \textsc{Ag}(\textit{List Val}))^{?})$ is also undefined. Hence no elements of the RA can be persistent – not even the fragmental ones.

With this, we may now define $\gamma \Mapsto_\bullet xs_v$ and $\gamma \Mapsto_\circ xs_v$ formally.

Definition 8.2.1 (Authoritative Abstract State)**.** $\gamma \Mapsto_\bullet xs_v \triangleq \boxed{\bullet\,(1, \mathsf{ag}\ xs_v)}^{\gamma}$

Definition 8.2.2 (Fragmental Abstract State)**.** $\gamma \Mapsto_\circ xs_v \triangleq \boxed{\circ\,(1, \mathsf{ag}\ xs_v)}^{\gamma}$

Lemma 1 (Abstract State Alloc)**.** For any abstract state xs_v, we have

$$\vdash \Rrightarrow \exists \gamma.\ \gamma \Mapsto_\bullet xs_v * \gamma \Mapsto_\circ xs_v$$

Proof. By Own-op, it suffices to show $\Rightarrow\!\exists\gamma.\boxed{\bullet(1, \mathsf{ag}\ xs_v) \cdot \circ(1, \mathsf{ag}\ xs_v)}^{\gamma}$. This follows by Ghost-alloc if we can show that the element is valid. That is, $\bullet(1, \mathsf{ag}\ xs_v) \cdot \circ(1, \mathsf{ag}\ xs_v) \in \mathcal{V}$. However, this follows by the definitions of the involved resource algebras. □

In section 3.5 we postulated two important lemmas which showed that the authoritative and fragmental states agree, and how to update the abstract state. We now proceed to prove these lemmas.

Lemma 2 (Abstract State Agree)**.** For a ghost name γ and abstract states xs_v and xs'_v, we have

$$\gamma \Rightarrow\!\!\!\bullet\ xs'_v * \gamma \Rightarrow\!\!\!\circ\ xs_v \vdash xs_v = xs'_v$$

Proof. Since we own both $\gamma \Rightarrow\!\!\!\bullet\ xs_v$ and $\gamma \Rightarrow\!\!\!\circ\ xs'_v$, and they have the same ghost name, we can use rules Own-op and Own-valid to conclude that the element $\bullet(1, \mathsf{ag}\ xs_v) \cdot \circ(1, \mathsf{ag}\ xs'_v)$ is valid. By the authoritative RA, this means that $\mathsf{ag}\ xs'_v \preccurlyeq \mathsf{ag}\ xs_v$. By definition of the agreement RA, this means that $xs'_v = xs_v$, which is what we wanted. □

Lemma 3 (Abstract State Update)**.** For any ghost name γ, and abstract values xs_v, xs'_v, and xs''_v, we have

$$\gamma \Rightarrow\!\!\!\bullet\ xs'_v * \gamma \Rightarrow\!\!\!\circ\ xs_v \Rrightarrow \gamma \Rightarrow\!\!\!\bullet\ xs''_v * \gamma \Rightarrow\!\!\!\circ\ xs''_v$$

Proof. This time, we use rule Own-op to conclude $\boxed{\bullet(1, \mathsf{ag}\ xs_v) \cdot \circ(1, \mathsf{ag}\ xs'_v)}^{\gamma}$. Further, by Own-op, it suffices to prove $\Rightarrow\!\boxed{\bullet(1, \mathsf{ag}\ xs''_v) \cdot \circ(1, \mathsf{ag}\ xs''_v)}^{\gamma}$. We do this by applying the Ghost-update rule, which requires us to prove $\bullet(1, \mathsf{ag}\ xs_v) \cdot \circ(1, \mathsf{ag}\ xs'_v) \rightsquigarrow \bullet(1, \mathsf{ag}\ xs''_v) \cdot \circ(1, \mathsf{ag}\ xs''_v)$. Firstly by the product, fractional, and agreement RA, the element $(1, \mathsf{ag}\ xs''_v)$ is valid, so the product of the authoritative and fragmental parts of it is also valid. Next, note that we own the entire fraction of the fragmental element. Hence, there can be no other valid fragments. It hence follows that we can do the frame-preserving update. □

8.3 Abstract Points-to and Reachability

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

Definition 8.3.1 (Abstract Reach)**.** $x \dashrightarrow \gamma \triangleq \boxed{\circ\{x\}}^{\gamma}$

Definition 8.3.2 (Abstract Points-to)**.** $\gamma \rightarrowtail x \triangleq \exists s.\boxed{\bullet s}^{\gamma} * \mathbin{\text{✳}}_{x_m \in s} x_m \rightsquigarrow x$

One should think of sets $s \in \mathcal{P}(Node)$ 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 a fragmental element s and authoritative element t, we get that the fragmental element is "smaller" than the authoritative, $s \preccurlyeq t$. 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 in section 7.2. We proceed to prove the four essential lemmas on abstract points-to and reachability that we postulated in said section.

Lemma 20 (Abs Reach Alloc)**.** For all nodes x, we have

$$x \rightsquigarrow x \Rrightarrow (\exists\gamma.\ \gamma \rightarrowtail x * x \dashrightarrow \gamma)$$

Proof. We assume $x \rightsquigarrow x$ and must show $\Rightarrow\!\exists\gamma'.\ \gamma' \rightarrowtail x * x \dashrightarrow \gamma'$. By definition or the authoritative RA, the element $\bullet\{x\} \cdot \circ\{x\}$ is valid. Hence by the Ghost-alloc rule, we may get $\Rightarrow\!\exists\gamma.\boxed{\bullet\{x\} \cdot \circ\{x\}}^{\gamma}$. By the upd-mono rule, 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 rule 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. □

Lemma 21 (Abs Reach Concr)**.** For nodes x_n and x_m and ghost names γ_m, we have

$$x_n \dashrightarrow \gamma_m * \gamma_m \rightarrowtail x_m \mathrel{-\!\!*} x_n \rightsquigarrow x_m * \gamma_m \rightarrowtail x_m$$

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 $\ast_{x' \in s} x' \rightsquigarrow x_m$. 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 $x_n \in s$. Note that this does not consume $\boxed{\bullet s}^{\gamma_m}$ as that assertion is persistent. This allows us to frame away the second part of the goal, $\gamma_m \rightarrowtail x_m$, using $\boxed{\bullet s}^{\gamma_m}$ and $\ast_{x' \in s} x' \rightsquigarrow x_m$. As the latter of these assertions is persistent, we still own it. 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. □

Lemma 22 (Abs Reach Abs)**.** For nodes x_n and x_m and ghost names γ_m, we have

$$x_n \rightsquigarrow x_m * \gamma_m \rightarrowtail x_m \Rrightarrow x_n \dashrightarrow \gamma_m * \gamma_m \rightarrowtail x_m$$

Proof. Assuming $x_n \rightsquigarrow x_m$ and $\gamma_m \rightarrowtail x_m$ we must conclude $\Rrightarrow x_n \dashrightarrow \gamma_m$. From $\gamma_m \rightarrowtail x_m$ we know that there is some set s so that $\boxed{\bullet s}^{\gamma_m}$ and $\ast_{x' \in s} x' \rightsquigarrow x_m$. 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 $\mathcal{P}(Node)$ 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 $\ast_{x' \in \{x_n\} \cup s} x' \rightsquigarrow x_m$. However, by $\ast_{x' \in s} x' \rightsquigarrow x_m$ and our assumption that $x_n \rightsquigarrow x_m$, we can easily conclude this.

□

Lemma 23 (Abs Reach Advance)**.** For nodes x_m and x_o and ghost names γ_m, we have

$$\gamma_m \rightarrowtail x_m * x_m \rightsquigarrow x_o \Rrightarrow \gamma_m \rightarrowtail x_o * x_o \dashrightarrow \gamma_m$$

Proof. Assuming $\gamma_m \rightarrowtail x_m$ and $x_m \rightsquigarrow x_o$ we must prove $\Rrightarrow \gamma_m \rightarrowtail x_o * x_o \dashrightarrow \gamma_m$. From $\gamma_m \rightarrowtail x_m$, we get some set s so that $\boxed{\bullet s}^{\gamma_m}$ and $\ast_{x' \in s} x' \rightsquigarrow x_m$. As we did in the proof of lemma 22, 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: $\boxed{\circ \{x_o\}}^{\gamma_m}$, 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 $\ast_{x' \in \{x_o\} \cup s} x' \rightsquigarrow x_o$. We already know that $\ast_{x' \in s} x' \rightsquigarrow x_m$ and $x_m \rightsquigarrow x_o$. Thus, by transitivity of reach (lemma 16), we may conclude $\ast_{x' \in \cup s} x' \rightsquigarrow x_o$. Thus, we are done if we can prove that $x_o \rightsquigarrow x_o$, which by lemma 15 amount to showing that x_o is a node. However, since $x_m \rightsquigarrow x_o$, then, by lemma 18, we know that this is the case. □

Chapter 9

Conclusion and Future Work

In this project, we gave three queue specifications, expressed in the Iris program logic. The *sequential specification* allows client to track the contents of the queue, but the queue cannot be used concurrently. The *concurrent specification* allows for concurrency, but there is no functionality for tracking the queue contents. The final and most general specification – the *Hocap-style specification* – can not only be used by concurrent clients, it also allows clients to track the queue contents. This functionality was made possible by introducing suitable predicates which track the queue contents. In particular, we introduced two "views" of the abstract state of the queue, xs_v: the *authoritative* view $\gamma \Rightarrow_\bullet xs_v$, and the *fragmental* view $\gamma \Rightarrow_\circ xs_v$. The authoritative view is owned by the queue, and the fragmental view is owned by clients. The construction of the predicates enforces that the views agree on the state of the queue and ensures they can only be updated in unison. This gives clients a way to track the contents of the queue. We furthermore demonstrated that the sequential and concurrent specifications are derivable from the Hocap-style specification. That is, a queue satisfying the Hocap-style specification also satisfies the other two.

We gave an implementation of the Two-Lock M&S Queue in HeapLang, which essentially consists of a linked list, making up the elements of the queue, and a *head* and *tail* pointer, pointing to nodes in the linked list. The head pointer marks the start of the queue and is used in dequeueing, and the tail pointer is near the end of the queue and is used for enqueuing new values. We demonstrated the possibility of the tail lagging behind the head – a situation thought to be impossible. However, even with this, we were able to show that the queue adheres to all three specifications. Most notably, proving it satisfies the Hocap style specification was done by establishing a queue invariant, which identified the four possible states the queue could be in, and what the ownership of the queue resources looked like in each of the four states. The proof then showed how instructions in the code changed the state of the queue, and which queue resources participating threads were allowed to own in each state. In particular, we used the fact that we can split points to predicates to allow e.g. an enqueueing thread to own $\ell_{\text{head}} \mapsto^{\frac{1}{2}} \text{in}(x_{\text{head}})$ while enqueueing. This assures the thread that the head pointer cannot be updated while it is working on enqueueing its value.
The invariant further maintained the relationship between the concrete state of the queue, represented by the underlying linked list, and the abstract state of the queue, represented by the aforementioned predicate, $\gamma \Rightarrow_\bullet xs_v$. This allowed us to show how the queue functions, enqueue and dequeue, affect the contents of the queue, which in turn gives clients a way to track the contents of the queue.
Having the queue resources in an invariant allows us to support concurrent clients as invariants are duplicable in Iris, which means that the invariant and its resources can be shared between threads.

Similarly, we gave an implementation of the Lock-Free M&S Queue in HeapLang and proved its compliance with the Hocap-style specification. Unlike the Two-Lock M&S Queue, the Lock-Free M&S Queue *does* rely on the fact that the tail does not lag behind the head. It further relies on the fact that the head and tail only ever move towards the end of the linked list. The queue invariant we created for the Lock-Free M&S Queue thus incorporated these two characteristics. We used the notion of *reachability* introduced in Vindum and Birkedal [2021] to describe the properties formally. The concrete reachability predicate, $x \rightsquigarrow x'$ denotes that node x precedes node x' in the linked list making up the queue. The abstract points-to predicate, $\gamma \rightarrowtail x$, allows us to keep track of the location of specific types of nodes, such as the head node, the tail node, and the last node, which can change during the course of the queues

lifetime. Together with the abstract reachability predicate, $x \dashrightarrow \gamma$, we could conclude that if a node could concretely reach, say, the head node, then it could concretely reach any future head nodes. The abstract points-to and reachability predicates we put in the invariant then allowed us to conclude that the head can *always* concrete reach the tail.
Finally, this queue invariant also maintained the same relationship between the concrete and abstract states of the queue, which enabled accurate tracking of the queue contents.

►Future work◄ ►Possibility of simplifying queue invariant for lock-free by removing isLL (and adding x_last -¿ None)◄ ►Proving correctness of a queue client◄ ►The two queues studied in this project are somewhat related, so it would be interesting to see if other queues also adhere to the specifications given.◄

Bibliography

Lars Birkedal and Aleš Bizjak. Lecture notes on iris: Higher-order concurrent separation logic. `http://iris-project.org/tutorial-pdfs/iris-lecture-notes.pdf`, 2017.

Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. doi: 10.1145/78969.78972. URL `https://doi.org/10.1145/78969.78972`.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28:e20, 2018. doi: 10.1017/S0956796818000151. URL `https://doi.org/10.1017/S0956796818000151`.

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. doi: 10.1145/248052.248106. URL `https://doi.org/10.1145/248052.248106`.

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. doi: 10.1145/3437992.3439930. URL `https://doi.org/10.1145/3437992.3439930`.

Appendix A

Common Definitions and Lemmas

A.1 Lists and Nodes

Definition A.1.1 (First in Queue)**.** $\text{isFirst}(x, xs) \triangleq \exists xs_{rest}.\, xs = xs_{rest} \mathbin{+\!\!+} [x]$

Definition A.1.2 (Last in Queue)**.** $\text{isLast}(x, xs) \triangleq \exists xs_{rest}.\, xs = x :: xs_{rest}$

Definition A.1.3 (Second Last in Queue)**.** $\text{isSndLast}(x, xs) \triangleq \exists x_{last}, xs_{rest}.\, xs = x_{last} :: x :: xs_{rest}$

Lemma 28 (Adding/Removing Non-Last)**.** $\forall x, y, xs, ys.$
$\text{isLast}(x, (xs \mathbin{+\!\!+} [y] \mathbin{+\!\!+} ys)) \iff \text{isLast}(x, (xs \mathbin{+\!\!+} [y]))$

Lemma 29 (List Destruction Equality)**.** $\forall xs_1, x_1, xs_2, x_2.$
$xs_1 \mathbin{+\!\!+} [x_1] = xs_2 \mathbin{+\!\!+} [x_2] \implies xs_1 = xs_2 \wedge x_1 = x_2$

Definition A.1.4 (Value Projection)**.**

$$\begin{aligned} \text{projVal}([]) &\triangleq [] \\ \text{projVal}(x :: xs) &\triangleq \text{val}(x) :: \text{projVal}(xs) \end{aligned}$$

Lemma 30 (Value Projection Split)**.** $\forall xs_1, xs_2.$
$\text{projVal}(xs_1 \mathbin{+\!\!+} xs_2) = \text{projVal}(xs_1) \mathbin{+\!\!+} \text{projVal}(xs_2)$

Definition A.1.5 (Wrap Some)**.**

$$\begin{aligned} \text{wrapSome}([]) &\triangleq [] \\ \text{wrapSome}(x :: xs) &\triangleq \text{Some}\, x :: \text{wrapSome}(xs) \end{aligned}$$

Lemma 31 (Wrap Some Split)**.** $\forall xs_1, xs_2.$
$\text{wrapSome}(xs_1 \mathbin{+\!\!+} xs_2) = \text{wrapSome}(xs_1) \mathbin{+\!\!+} \text{wrapSome}(xs_2)$

Definition A.1.6 (The "All" Predicate)**.**

$$\begin{aligned} \text{All}([], \Psi) &\triangleq \mathsf{True} \\ \text{All}(x :: xs, \Psi) &\triangleq \Psi(x) * \text{All}(xs) \end{aligned}$$

Lemma 32 (All Split)**.** $\forall xs_1, xs_2, \Psi.$
$\text{All}(xs_1 \mathbin{+\!\!+} xs_2, \Psi) \mathbin{**} \text{All}(xs_1, \Psi) \mathbin{+\!\!+} \text{All}(xs_2, \Psi)$

Lemma 33 (Node Equality)**.**

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

A.2 Results About the isLL Predicate

Lemma 34 (Extract Chain from isLL)**.** $\forall xs.\ \mathrm{isLL}(xs) \mathrel{-\!\!*} \mathrm{isLL}(xs) * \mathrm{isLL\_chain}(xs)$

Lemma 35 (isLL_chain Nodes)**.** $\forall xs_1, x, xs_2.$
$\mathrm{isLL\_chain}(xs_1 \mathbin{+\!\!+} [x] \mathbin{+\!\!+} xs_2) \mathrel{-\!\!*} \mathrm{in}(x) \mapsto^{\square} (\mathrm{val}(x), \mathrm{out}(x))$

Lemma 36 (isLL_chain Split)**.** $\forall xs, ys.$
$\mathrm{isLL\_chain}(xs \mathbin{+\!\!+} ys) \mathrel{-\!\!*} \mathrm{isLL\_chain}(xs) * \mathrm{isLL\_chain}(ys)$

Lemma 37 (isLL Split)**.** $\forall xs, ys.$
$\mathrm{isLL}(xs \mathbin{+\!\!+} ys) \mathrel{-\!\!*} \mathrm{isLL}(xs) * \mathrm{isLL\_chain}(ys)$