Verifying a Single-Enqueuer Single-Dequeuer Wait-Free Queue in Iris

Freja Marott Crawford, 201908608

Master's Thesis

Advisor: Amin Timany

Department of Computer Science, Aarhus University

15/6/2026

AARHUS UNIVERSITET

Abstract

Reasoning about concurrent programs is challenging due to shared mutable state and interference between threads. This becomes particularly difficult when programs explicitly free memory, since one thread may access a location that another thread has already freed.

In this thesis, we formally verify a wait-free single-enqueuer, single-dequeuer (SESD) queue algorithm by Jayanti and Petrovic [JP05] in the Iris separation logic framework using the Rocq proof assistant [The26]. The SESD queue is a building block of the main queue algorithm presented in [JP05]: a wait-free multiple-enqueuer, single-dequeuer (MESD) queue algorithm that runs in logarithmic time. The SESD queue algorithm consists of four operations: enqueue, dequeue, and two variants of readFront. We first implement the queue operations in OCaml and then in HeapLang, a language designed for verification in Iris.

We verify the SESD queue algorithm in three versions that build on top of each other:
First, we prove a sequential specification of the SESD queue algorithm to establish correctness of the queue operations. Second, we verify a concurrent specification of a simplified version of the algorithm, where no explicit memory freeing is performed. Here we focus on defining an appropriate Iris invariant for a duplicable queue predicate that can be shared between the enqueuer and the dequeuer threads. In particular, we are inspired by the notions of *reachability* and *abstract reachability*, which were introduced in connection with verification of the Michael-Scott queue in [VB21]. Finally, we verify the full concurrent SESD algorithm with explicit memory freeing. The main challenge in this setting is reasoning about locations that are safe to read and safe to free. We introduce a logical alternative to points-to assertions, and we keep track of which parts of memory have been freed. In addition, we add extra ghost state to enable communication between enqueuer and dequeuer threads. Using these tools, we can prove that the dequeuer never frees a part of memory that the enqueuer may access.

Overall, in this thesis, we verify the SESD queue algorithm in Rocq using Iris in a sequential version, a concurrent version without memory freeing, and a concurrent version with explicit memory freeing. This work supports future verification of the full MESD queue algorithm.

Resumé

Det er udfordrende at ræsonnere omkring programmer, der køres parallelt af flere forskellige tråde. Det skyldes især interferens mellem trådene, samt at hukommelsen kan tilgås og ændres af alle trådene. Dette bliver endnu vanskeligere, når der eksplicit frigøres hukommelse i programmer, idet én tråd kan forsøge at tilgå hukommelse, som en anden tråd har frigjort.

I dette speciale verificerer vi formelt en *wait-free single-enqueuer, single-dequeuer* (SESD) køalgoritme af Jayanti og Petrovic [JP05] med Iris separationslogik-frameworket i bevisassistenten Rocq [The26]. SESD-køen er en byggesten i den egentlige køalgoritme, der præsenteres i [JP05]: En *wait-free multiple-enqueuer, single-dequeuer* (MESD) køalgoritme, der kører i logaritmisk tid. SESD kø-algoritmen består af fire operationer: `enqueue`, `dequeue` og to varianter af `readFront`. Vi implementerer først køoperationerne i OCaml og derefter i HeapLang, der er designet til at verificere programmer med Iris.

Vi verificerer SESD kø-algoritmen i tre versioner, der bygger ovenpå hinanden.
Først beviser vi en sekventiel specifikation af SESD køalgoritmen for at bevise korrektheden af køoperationerne. Dernæst verificerer vi en parallel specifikation af en forsimplet version af algoritmen, hvor der ikke eksplicit frigøres hukommelse. Her fokuserer vi på på at definere en passende Iris-invariant til et duplikerbart køprædikat, som kan deles mellem enqueuer- og dequeuer-trådene. Vi er især inspireret af begreberne *reachability* og *abstract reachability*, der blev introduceret i forbindelse med verifikation af Michael-Scott-køalgoritmen i [VB21]. Til slut verificerer vi den komplette parallelle SESD køalgoritme, der inkluderer eksplicit hukommelsesfrigørelse. Den største udfordring i denne kontekst er at ræsonnere om, hvornår hukommelse er sikkert at tilgå og at frigøre. Vi introducerer et logiske alternativ til *points-to assertions*, og vi holder styr på, hvilke dele af hukommelsen, der er blevet frigjort. Derudover tilføjer vi yderligere *ghost state* for at muliggøre kommunikation mellem enqueuer- og dequeuer-trådene. Med disse værktøjer kan vi bevise, at dequeuer-tråden aldrig frigør en del af hukommelsen, som enqueuer-tråden kan forsøge at tilgå.

Alt i alt verificerer vi i dette speciale SESD-køalgoritmen i Rocq med Iris i en sekventiel version, en parallel version uden at frigøre hukommelse, og i en parallel version, der inkluderer eksplicit hukommelsesfrigørelse. Dette arbejde understøtter eventuel fremtidig verifikation af MESD køalgoritmen.

Contents

1 Introduction

Reasoning about concurrent programs is challenging due to interference between threads and shared memory, where small errors in the implementation can lead to race conditions and violations of safety properties. This motivates the use of formal verification, which can ensure correctness and safety of programs.
Iris is a higher-order concurrent separation logic framework that enables modular reasoning about concurrent programs, such as shared data structures. Iris has previously been used to verify a range of concurrent algorithms and data structures, for example the Michael-Scott queue [VB21, Ped24].

In this thesis, we use the Iris separation logic framework [JKJ+18] in the Rocq proof assistant [The26] to formally verify a wait-free queue algorithm in a concurrent setting. The algorithm we study is the single-enqueuer, single-dequeuer (SESD) queue, which is a fundamental part of the multiple-enqueuer, single-dequeuer (MESD) logarithmic-time wait-free queue algorithm presented by Jayanti and Petrovic [JP05]. In the MESD algorithm, each enqueuer works on a local SESD queue and collaborates with the other threads to propagate the front elements of these local queues to the root of a shared binary tree. As a result, the correctness of the MESD algorithm relies on the correctness of the SESD queues. The goal of this work is to formally verify the SESD algorithm in Rocq, including reasoning about explicit freeing of memory, as it appears in Jayanti and Petrovic [JP05]. In particular, we define and prove specifications for the queue operations in increasingly complicated settings. The algorithm consists of four operations: The usual queue operations, enqueue and dequeue, and two auxiliary operations used in the MESD algorithm: two variants of readFront, one for the enqueuer and one for the dequeuer.

We verify the SESD queue algorithm in three settings, each building on the previous one.

1. We prove a sequential specification of the SESD algorithm. This setting focusses on building an understanding of the queue algorithm and reasoning about a queue in Iris.

2. We prove a concurrent specification for a simplified version of the algorithm in which we do not free memory. This setting allows us to focus on the concurrency between the enqueuer and dequeuer. We make the queue predicate duplicable using an Iris invariant, distribute points-to assertions between the two threads, and use ghost state to capture how the queue changes during the executions of concurrent queue operations.

3. We prove a concurrent specification for the full SESD algorithm, including freeing memory. This requires additional reasoning about when a location is safe to read for the enqueuer and safe to free for the dequeuer. This coordination between the enqueuer and dequeuer is captured using additional ghost state in the queue invariant.

The algorithm is originally presented in pseudocode in Jayanti and Petrovic [JP05]. We first implement the algorithm in OCaml and then translate it into HeapLang, a language designed for verification in Iris. This translation from the paper's pseudocode to HeapLang via OCaml allows us to have executable and testable programs before verifying them.

All proofs are mechanised in Rocq.

The report is organised as follows: Section 2 presents the paper's [JP05] queue algorithm, focussing on the single enqueuer, single dequeuer (SESD) version. In section 3, we give a brief overview of the Iris framework and HeapLang. Section 4 discusses related work on the verification of similar data structures in Iris. Section 5 presents the HeapLang implementation of the four operations in the SESD queue algorithm. In section 6 we define and prove the sequential specification. In section 7, we prove the concurrent specification without freeing. We focus on how we adapt the queue predicate from the sequential to the concurrent setting using Iris invariants. Finally, in section 8, we prove the concurrent specification of the full SESD algorithm including freeing. This introduces many additional ghost resources to the queue invariant compared to the non-freeing version. We focus on why these additions are necessary in the proofs of the specification. In section 9 we conclude and discuss future work, including a higher-order concurrent abstract predicate (HOCAP) specification of the SESD algorithm, and a verification of the full MESD algorithm.

2 A Sub-Linear Wait-Free Queue Algorithm

In the paper *Logarithmic-Time Single Deleter, Multiple Inserter Wait-Free Queues and Stacks* [JP05], Jayanti and Petrovic present a wait-free queue algorithm with multiple enqueuers and a single dequeuer. The authors additionally present a single-enqueuer, single-dequeuer (SESD) version of the algorithm, that acts as a building-block of the full multiple-enqueuer, single-dequeuer (MESD) queue algorithm.

The authors highlight two main properties of the MESD queue algorithm:

1. **Sub-linear time complexity:** both enqueue and dequeue operations run in $O(\log n)$ worst-case time.
2. **Space efficiency:** the algorithm uses $O(n + m)$ space, where n is the number of threads and m is the number of stored elements.

2.1 Single Enqueuer, Single Dequeuer Queue

We focus on the single-enqueuer, single-dequeuer (SESD) version of the algorithm.
The SESD queue algorithm has four operations: enqueue, dequeue, and two versions of readFront that reads the front element of the queue; one for the enqueuer and one for the dequeuer. We will distinguish them by naming them readFrontEnq and readFrontDeq.

Below, we define the types, the shared variables and the initialisation of the shared variables as described in the paper.

Types:

* NodeType = record { v : Val[1], next : ↑ NodeType }

Shared Variables:

* First, Last, Announce, FreeLater : ↑ NodeType
* Help : Val

Initialisation:

* First, Last := new Node();
* FreeLater := new Node();

The queue is represented as a linked list of nodes: A node is a record consisting of a value (a queue element) and a pointer to the node in the linked list, which contains the next queue element.
There are five shared variables between the dequeuer and the enqueuer; The First and Last pointers are used to locate the first and last nodes of the linked list (the front and back elements of the queue), while the rest of the shared variables – Announce, FreeLater and Help – are auxiliary variables used to ensure safety of the queue algorithm.
The queue is initialised by letting First and Last locations point to the same empty node, the *dummy node* of the linked list, and letting the FreeLater location point to a different empty node. This step ensures that there is always a node to free.

Based on these types and the shared variables, we can illustrate a (non-empty) queue as a linked list as follows:

[1] The type of the elements in the queue can be freely chosen by the client.

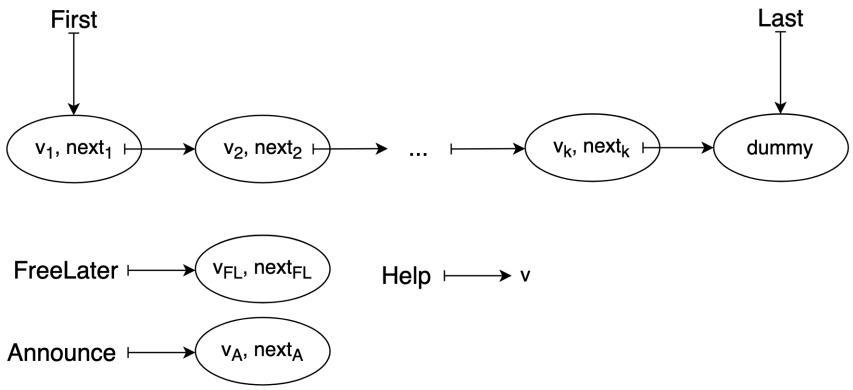

Figure 1: Illustration of the queue and the shared variables. Each v_i is a queue element, and each next_i is a location pointing to the next node.

The nodes pointed to by FreeLater and Announce are part of the linked list, and the value stored in Help is the value of the current or the previous front node. We denote the last node of the linked list the *dummy node*, since it does not contain any queue element.

We go through the four operations as described in the paper's [JP05] pseudocode. The pseudocode is very high-level and leaves many operations implicit. For example, every time a pointer is mentioned in the pseudo-code, it is dereferenced. When we e.g. compare `tmp == Last`, we dereference the Last pointer, and compare the contents of the nodes.

2.1.1 Enqueue

To enqueue a value v to the queue, we first create a new empty (dummy) node, `newNode`. Second, we read the current last node (the dummy node) `tmp` and write the value v into it. Next, we insert a the pointer `newNode` to the newly created node into `tmp`, such that the freshly allocated node `newNode` becomes the new dummy node of the linked list. Finally, we move the Last pointer to point to the dummy node.

Pseudocode (from [JP05]):

```
 function enqueue(v : Val) : void {
1    newNode = new Node();
2    tmp = Last;
3    tmp.val = v;
4    tmp.next = newNode;
5    Last = newNode;
 }
```

We illustrate the enqueue operation and its effect on the linked list:

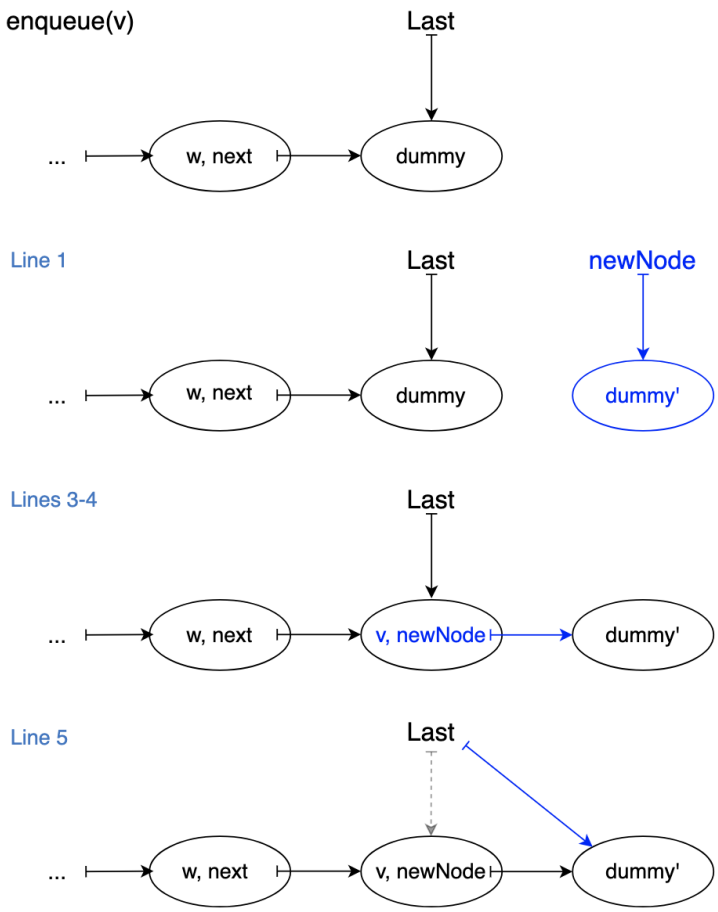

Figure 2: Enqueuing a value v.

2.1.2 Dequeue

To dequeue a value from the queue, we first check if the queue is empty by checking if the First pointer is equal to the Last pointer. If it is empty, we return $\perp$. Otherwise, we read the current front node in the queue, `tmp`, and move the First pointer to point to the the second node in the queue.
We check whether the front node `tmp` has been announced by the enqueuer. If that is the case, we free the node stored in FreeLater, and store the announced front node `tmp` in FreeLater.
If the node `tmp` has *not* been announced, we free it.
In both cases, we return the value stored in the front node `tmp`.

Pseudocode (from [JP05]):

```
  function dequeue() : Val {
6   tmp = First;
7   if (tmp == Last) { return ⊥; }
8   retVal = tmp.val;
9   Help = retVal;
10  First = tmp.next;
11  if (tmp == Announce) {
12      tmp' = FreeLater;
13      FreeLater = tmp;
14      free(tmp');
     } else {
15      free(tmp);
     }
16  return retVal;
  }
```

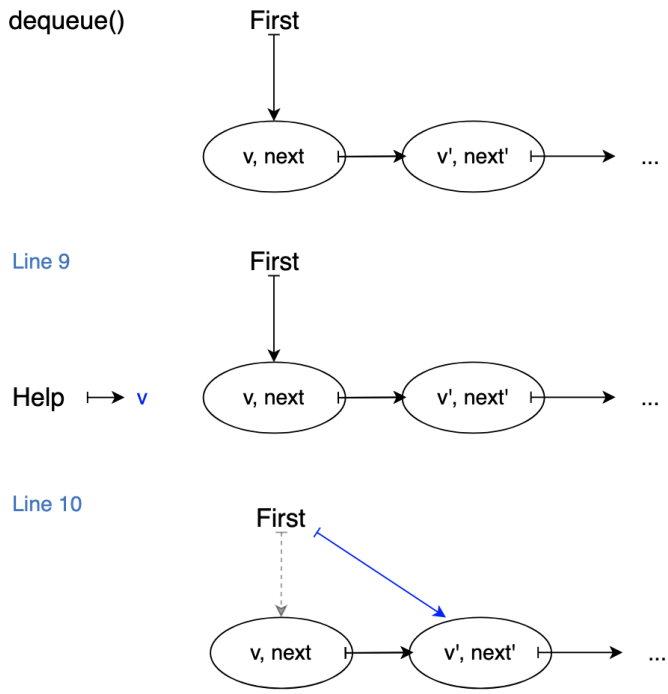

Figure 3: First part of dequeue: Store the value v in `Help` and move the First pointer.

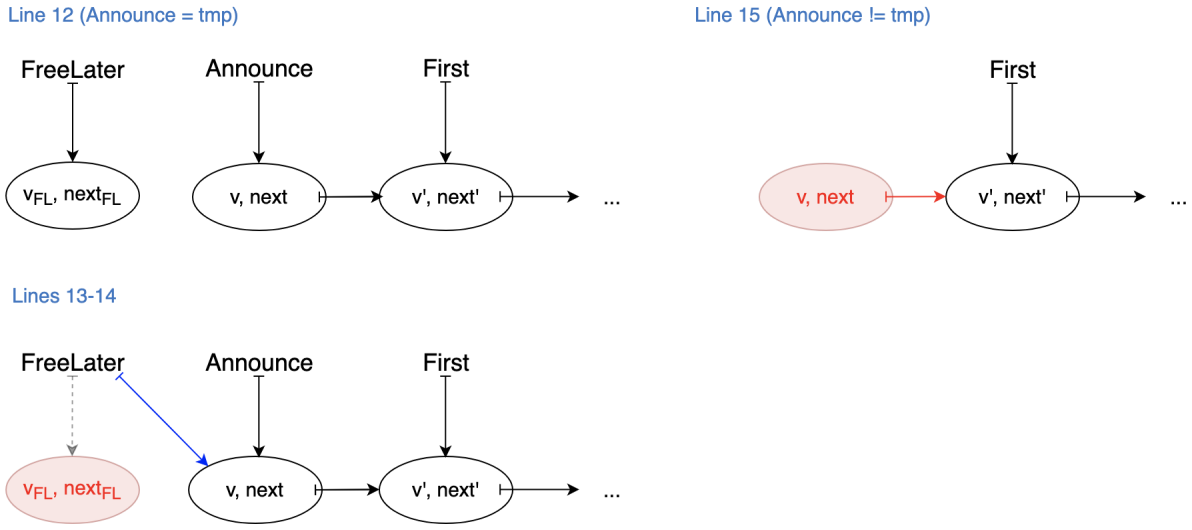

Figure 4: Second part of dequeue: If the current first node, `tmp`, has been announced, we free the node that FreeLater points to, and mark `tmp` to be freed later. Otherwise, we free `tmp`.

2.1.3 readFront (Enqueuer)

We first read the front element and store it in `tmp`. If the queue is empty, we return $\perp$. Otherwise, we announce that we will read the value stored in `tmp` by storing `tmp` in shared Announce variable. This signals to the dequeuer that it should *not* free the memory used by this node yet.
We then read the variable First again to check if it has changed since we last read it, i.e. if the current first is still `tmp`.
If First has changed, it is not necessarily safe to read `tmp`. Instead, we read and return the value stored in the shared variable Help. The Help variable is updated in the dequeue operation, and thus either contains the value of the current front element of the queue, or the value of some element that has been the front element within this ReadFront operation.

If First has not changed, then it is safe to read and return the value stored in node tmp.

Pseudocode (from [JP05]):

```
  function readFrontEnq() : Val {
17    tmp = First;
18    if (tmp == Last) { return ⊥; }
19    Announce = tmp;
20    if (tmp ≠ First) {
21        retVal = Help;
      } else {
22        retVal = tmp.val;
      }
23    return retVal;
  }
```

2.1.4 readFront (Dequeuer)

The readFront operation for the dequeuer is straightforward. We simply check whether the queue is empty, in which case we return $\bot$, or we return the value stored in the front element of the queue. This is always safe, since only the dequeuer itself can free the first node.

Pseudocode (from [JP05]):

```
  function readFrontDeq() : Val {
24    tmp = First;
25    if (tmp == Last) { return ⊥; }
26    return tmp.val;
  }
```

2.2 Multiple Enqueuer, Single Dequeuer

In the paper, the single-enqueuer, single-dequeuer (SESD) algorithm is designed as a building-block of the main queue algorithm, the multiple enqueuer, single dequeuer version (MESD).
In this setting, we have a single dequeuer and n concurrent enqueuers. Each enqueuer is assigned a unique thread id $tid \in \{1, \ldots, n\}$ and maintains its own local SESD queue (described in section 2.1). Importantly, each enqueuer can only interact with its own local queue.

The main idea of the MESD queue algorithm is to keep a shared binary tree, where each internal node represents a local queue and its front element. Concretely, it contains a pair (ts, tid) of the thread id of an enqueuer and the timestamp of the front element of that enqueuer's local SESD queue. The invariant of the tree ensures that the timestamp ts is the smallest timestamp observed in its subtree. The n leaves correspond to the local queues of the n enqueuers. All threads propagate minimum timestamps bottom-up towards the root of the tree in each operation, which ensures that the root always contains the pair (ts_{min}, tid) with the globally minimal timestamp, and thus identifies the front element of the global queue. Therefore, to dequeue from the global queue, the dequeuer reads the tid of the root of the tree and performs the SESD-dequeue operation on the corresponding SESD queue.
The SESD-readFront operation is used to read the front element of the local queue during propagation in the tree, which happens during dequeueing and enqueueing in the global queue.

Overall, the MESD algorithm uses the local SESD queues to ensure wait-free enqueue and dequeue operations for each thread. The binary tree gives a logarithmic-time algorithm for identifying the front element of the global queue by propagating the queue with the earliest timestamp of its front element to the root. This combination results in a sub-linear wait-free multiple-enqueuer, single-dequeuer queue algorithm.

3 Iris & HeapLang

We briefly introduce the parts of Iris and HeapLang that are relevant for this project. We assume that the reader is already familiar with the Iris framework and the presentation in the lecture notes [BB23],

and therefore only fix notation and highlight the aspects that are used throughout the thesis.

Iris [JKJ+18] is a higher-order concurrent separation logic framework implemented in the Rocq proof assistant. It has already been used to verify several concurrent data-structures of varying complexity, for example the Michael-Scott queue [VB21, Ped24]. In particular, Iris supports ghost state and invariants, which enable modular reasoning about shared memory and interference between concurrent threads. With Iris, we can prove safety and partial correctness of concurrent programs.

HeapLang is an untyped ML-like language with references and concurrency. It is designed as an example language for the Iris program logic and is thus a suitable language for verifying programs in Iris. We use the HeapLang language as it is defined in the Rocq formalisation of Iris.

In this project, we verify the SESD queue algorithm using an abstract specification style, where the queue predicate is existentially quantified and exposed to the client. For example, the sequential queue specification (section 6) existentially quantifies a queue predicate $\mathsf{isQueueSeq}$:

$$
\begin{aligned}
&\exists\, \mathsf{isQueueSeq} : \mathit{Val} \to \mathit{list}(\mathbb{N} \times \mathbb{N} \times \mathit{Val}) \to (\mathit{Val} \to \mathit{iProp}) \to \mathit{iProp},\\
&\forall \Phi.\ \{\mathsf{True}\}\ \mathsf{init}()\ \{\mathsf{v}.\ \mathsf{isQueueSeq}(v, \mathsf{nil}, \Phi)\}\\
&\wedge\\
&\forall q, \Phi, xs, ts, tid, w.\\
&\quad \{\mathsf{isQueueSeq}(q, xs, \Phi) * \Phi(w)\}\ \mathsf{enqueue}(q, ts, tid, w)\ \{v. \ulcorner v = () \urcorner * \mathsf{isQueueSeq}(q, xs +\!\!+ [(ts, tid, w)])\}\\
&\wedge\ \ldots\ .
\end{aligned}
$$

There are other ways of designing specifications, for example by sealing the implementation in a module or by using the higher-order concurrent abstract predicate (HOCAP) style specification. We return to this discussion in section 9.

In the remainder of this section, we will mainly fix the notation of the logic which we will be using in the rest of the thesis, as well as comment on the change we make for the Hoare triple concerning memory freeing.

3.1 The Logic

We assume familiarity with the basic Iris program logic as presented in [BB23]. Below, we summarise the notation and rules most frequently used in the proofs.

Points-to Assertions
We use the following notation for points-to relations:

$$
\begin{aligned}
\ell \mapsto v &\quad \text{(normal points-to)}\\
\ell \mapsto_{\square} v &\quad \text{(persistent points-to)}\\
\ell \mapsto_{1/2} v &\quad \text{(fractional points-to)}
\end{aligned}
$$

The following standard properties will be used repeatedly (and often implicitly):

Lemma 3.1. (Properties of points-to assertions)

$$
\textsc{PersPtPers}\ \frac{\ell \mapsto_{\square} v}{\square(\ell \mapsto_{\square} v)} \qquad
\textsc{PtToPersPt}\ \frac{\ell \mapsto v}{\ell \mapsto_{\square} v} \qquad
\textsc{PersPtAgree}\ \frac{\ell \mapsto_{\square} v \qquad \ell \mapsto_{\square} v'}{\ulcorner v = v' \urcorner} \qquad
\textsc{PtExcl}\ \frac{\ell \mapsto v \qquad \ell \mapsto v'}{\mathsf{False}}
$$

$$
\textsc{PtSplit}\ \frac{\ell \mapsto v}{\ell \mapsto_{1/2} v * \ell \mapsto_{1/2} v} \qquad
\textsc{PtCombine}\ \frac{\ell \mapsto_{1/2} v * \ell \mapsto_{1/2} v'}{\ell \mapsto v * \ulcorner v = v' \urcorner}
$$

We slightly strengthen the Hoare triple for the expression $\mathsf{free}(\ell)$. Instead of returning True in the postcondition, we return an exclusive assertion stating that the location has been freed:

$$\{\ell \mapsto v\}\ \mathsf{free}(\ell)\ \{\mathsf{freed}(\ell)\}.$$

This assertion satisfies the following exclusivity properties:

$$\frac{\ell \mapsto v \qquad \mathsf{freed}(\ell)}{\mathsf{False}}\ \textsc{FreedPtFalse} \qquad\qquad \frac{\mathsf{freed}(\ell) \qquad \mathsf{freed}(\ell)}{\mathsf{False}}\ \textsc{FreedFreedFalse}$$

Iris supports *ghost state*, i.e. logical resources that exist only in the proof and not in the program itself. Concretely, ghost state consists of elements of a resource algebra (RA) with a ghost name γ. Threads can own fragments of ghost state and update them according to the rules of the underlying RA. Throughout this project, we use ghost state to record information about the queue that is not explicitly represented in memory, for example historical information about announced locations, logical points-to assertions, and abstract reachability relations between locations in the queue. We introduce the relevant ghost resources and their associated RAs as they are used in the proofs, and we assume familiarity with common RAs such as the agreement, authoritative, fractional and exclusive RAs.

Finally, in the proof sketches given in this thesis, we will – for the most part – ignore later modalities and update modalities. Reasoning about these is largely orthogonal to the interesting parts of the proofs, and most of the propositions we work with are timeless.

4 Related work

The concurrent specification given in this thesis is inspired by the style of specifications commonly used in Iris, for example for a concurrent stack and bag in the Iris lecture notes [BB23]. These specifications are weak in the sense that they focus on safety under concurrency, and only guarantee that operations return values that were previously inserted, without enforcing e.g. FIFO behaviour.

Concurrent queue algorithms have previously been verified in Iris. For example, the Michael-Scott queue [MS96], a concurrent lock-free queue. Since the SESD queue [JP05] studied in this thesis is also represented as a linked list modified concurrently by enqueuers and dequeuers, the verification of the Michael-Scott queue is particularly relevant. In the paper *Contextual refinement of the Michael-Scott queue (proof pearl)* [VB21], the authors prove contextual refinement between the Michael-Scott queue and a coarse-grained queue using Iris and ReLoC. In this work, the authors introduce the notions of *reachability* and *abstract reachability* between queue nodes, as well as the persistent points-to assertion. In our proofs, we reuse these notions of *reachability* and *abstract reachability*, together with persistent points-to assertions. We use reachability to reason about how queue nodes relate as the enqueuer and dequeuer concurrently modify the linked list representing the queue. The Michael-Scott queue has also been verified in the Master's thesis *Verification of the Blocking and Non-Blocking Michael-Scott Queue Algorithms* [Ped24], where the author develops a higher-order concurrent abstract predicate (HOCAP) specification for both the blocking and non-blocking variant of the queue. Reachability and abstract reachability is also used in these proofs.

There are, however, differences between the Michael-Scott queue and the SESD queue: The Michael-Scott queue supports multiple concurrent threads, and exists in a blocking and non-blocking variant. The SESD queue, on the other hand, is wait-free, but only allows concurrency between a single enqueuer and a single dequeuer. Although both algorithms are based on linked lists, the slightly different representations mean that we have to adapt the definitions of *reachability* and *abstract reachability* to our setting. Furthermore, the SESD algorithm explicitly frees memory during the dequeue operation. Reasoning about freeing memory in a concurrent setting is complicated, and we must come up with new techniques to reason about it.

Finally, since the concurrent specification in this thesis is weak in the sense that it does not capture the queue's FIFO behaviour, future work includes proving a HOCAP-style specification, similar to [Ped24].

Jayanti and Petrovic's SESD queue algorithm has also recently been studied in the paper *Verifying wait-freedom for concurrent higher-order programs* [NBT26]. In this work, the authors focus on a liveness property of the algorithm: They prove *wait-freedom* of the SESD queue, using the Lawyer concurrent separation logic, which is built on top of Iris. In contrast, this thesis focuses on proving safety and partial correctness properties of the SESD queue algorithm.

5 HeapLang Implementation of the Queue Algorithm

In this section, we present the implementation of the SESD queue algorithm [JP05] in HeapLang. The algorithm is written in pseudocode in the paper and thus leaves many implementation details implicit, such as initialisation, data representation, pointer comparison and shared variables. We implement the algorithm in two variants, with and without explicit freeing of memory. We only show the full implementation that includes freeing memory here, but both versions are available in Rocq.
We first give an intermediate OCaml implementation of the queue operations that we use to ease the implementation in HeapLang.

5.1 OCaml implementation

To make the translation from the pseudocode in the paper to HeapLang easier, we implemented the SESD algorithm in OCaml, including a simple test. This allows us to execute the algorithm and perform basic tests. We can check that our translation from pseudocode behaves as expected, including the initialisation of the queue and the types of the variables, that are not well specified in the paper's description of the algorithm. Furthermore, OCaml is relatively close to HeapLang (pattern matching, heap allocation, pointer equality, etc.), so the translation from the executable OCaml implementation to HeapLang is easier than implementing the programs directly in HeapLang. As a result, the OCaml implementation acts as an intermediate "testable" implementation that helps us translate the paper's pseudocode programs into HeapLang programs as closely as possible.

5.2 Types and Shared Variables

HeapLang does not have types. We want to imitate the types and the structure of the queue described in the paper [JP05]. There are some differences:

1. We represent the global variables – First, Last, Announce, FreeLater and Help – as pointers. Consequently, we use an extra *intermediate location* between the nodes compared to the nodes described in the paper. The main reason for this is to enable pointer comparisons, e.g. First == Last. By adding an intermediate location ℓ, we can check whether First and Last both point to the same ℓ instead of comparing the nodes directly. Since we add an intermediate locations, we also have to free an extra location in the dequeue program.

2. We represent the shared queue Q as a persistent points-to assertion, pointing to a tuple of the shared variables: $Q \mapsto_{\square} (\mathsf{First}, \mathsf{Last}, \mathsf{Announce}, \mathsf{FreeLater}, \mathsf{Help})$. This enforces agreement on the shared variables between the threads. In the programs, we access a shared variable, e.g. Last by reading Q and projecting to the second element of the tuple.

3. We use explicit option types for the shared variables Announce and Help. Therefore, in the programs, we make additional checks to see if a variable is $\mathsf{inj}_1\ ()$ or $\mathsf{inj}_2\ (v)$. We also represent the different types of nodes (normal or dummy) using $\mathsf{inj}_1\ ()$ or $\mathsf{inj}_2\ (v, \mathsf{next})$ respectively, which also adds additional pattern matching to the programs.

4. In our HeapLang implementation, we do not have records, so instead we represent a node as a pointer to a pair consisting of a value and a location pointing to the next node. For compatibility with the full MESD algorithm (section 2.2), the value itself consists of a timestamp, a thread id and the actual value of the queue element, e.g. $v_i = (ts_i, tid_i, w_i)$.

With these definitions, we can illustrate a (non-empty) queue with intermediate locations as follows (compare to the paper's [JP05] queue, fig. 1):

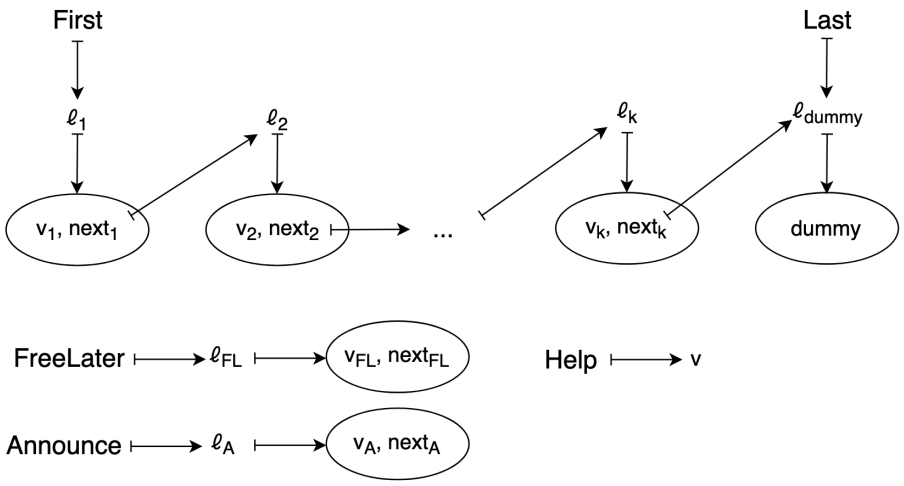

Figure 5: Illustration of the queue and the shared variables (HeapLang version).

5.3 Implementation

Below, we give the HeapLang implementation of the four operations. Compare with 2.1.

5.3.1 Initialisation

```
Definition init : val :=
  λ ().
    let dummyNode := ref inj1 () in
    let freeLaterNode := ref inj1 () in
    let First := ref dummyNode in
    let Last := ref dummyNode in
    let Announce := ref inj1 () in
    let FreeLater := ref freeLaterNode in
    let Help := ref inj1 () in
    ref (First, Last, Announce, FreeLater, Help).
```

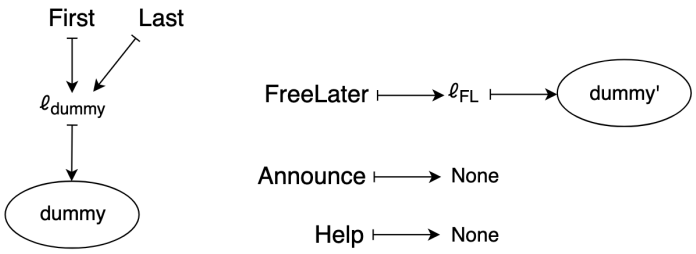

Figure 6: Illustration of the queue after initialisation.

5.3.2 Enqueue

```
Definition enqueue : val :=
  λ ts, tid, v, Queue.
    let newNode := ref inj1 () in
    let tmp := !((!Queue).Last) in
    tmp ← inj2 (ts, tid, v, ref newNode);;
    (!Queue).Last ← newNode.
```

5.3.3 Dequeue

```
Definition dequeue : val :=
  λ: Queue,
    let tmp := !(!Queue).First in
    if (tmp = !(!Queue).Last) then
        inj1 ()
    else
        match (!tmp) with
            inj1 () => assert False (* Impossible case *)
          | inj2 ts_tid_v_next =>
              (!Queue).Help ← inj2 (Fst ts_tid_v_next) ;;
              (!Queue).First ← !(Snd ts_tid_v_next) ;;
              if (inj2 tmp = !(!Queue).Announce) then
                  let FLL := !(!Queue).FreeLater in
                  (!Queue).FreeLater ← tmp ;;
                  match !FLL with
                      inj1 () => Free FLL
                    | inj2 FLL_v_next => Free FLL ;; Free (Snd FLL_v_next)
                  end ;; inj2 (Fst ts_tid_v_next)
              else
                  Free tmp ;;
                  Free (Snd ts_tid_v_next) ;;
                  inj2 (Fst ts_tid_v_next)
        end.
```

5.3.4 readFront (Enqueuer)

```
Definition read_front_enq : val :=
λ: Queue,
    let tmp := !(!Queue).First in
    if (tmp = !(! Queue).Last) then
        inj1 ()
    else
        ((!Queue).Announce) ← inj2 tmp ;;
        if (tmp ≠ (!(!Queue).First)) then
            !(! Queue).Help
        else
            match (!tmp) with
                inj1 () => assert False (* Impossible case *)
              | inj2 retVal => inj2 (Fst retVal)
            end.
```

5.3.5 readFront (Dequeuer)

```
Definition read_front_deq : val :=
λ: Queue,
    let tmp := !(!Queue).First in
    if (tmp = !(!Queue).Last) then
        inj1 ()
    else
        match (!tmp) with
            inj1 () => assert False (* Impossible case *)
          | inj2 retVal => inj2 (Fst retVal)
        end.
```

6 Sequential Specifications & Proofs

In this section, we first analyse the algorithm in a setting without concurrency, where the enqueuer and dequeuer interact with the queue sequentially. This simplified setting allows us to focus on the data

structures, variables and types used in the algorithm, and how we represent these in the logic. This includes examining how the shared variables and the linked list representing the queue are modified by each operation. Since we work in a setting without concurrency, the specification predicate isQueueSeq does not need to be persistent (duplicable): it is owned – and updated – exclusively by one thread at a time and is passed between the two threads in program-order.

6.1 Sequential Specification

$$
\begin{array}{l}
\exists\, \mathsf{isQueueSeq} : \mathit{Val} \to \mathit{list}(\mathbb{N} \times \mathbb{N} \times \mathit{Val}) \to (\mathit{Val} \to \mathit{iProp}) \to \mathit{iProp},\\
\forall \Phi.\ \{\mathsf{True}\}\ \ \mathsf{init}()\ \ \{\mathsf{v}.\ \mathsf{isQueueSeq}(v, \mathsf{nil}, \Phi)\}\\
\land\\
\forall q, \Phi, xs, ts, tid, w.\\
\quad \{\mathsf{isQueueSeq}(q, xs, \Phi) * \Phi(w)\}\ \ \mathsf{enqueue}(q, ts, tid, w)\ \ \{v. \ulcorner v = ()\urcorner * \mathsf{isQueueSeq}(q, xs +\!\!+ [(ts, tid, w)])\}\\
\land\\
\forall q, \Phi, xs.\\
\quad \{\mathsf{isQueueSeq}(q, xs, \Phi)\}\\
\qquad \mathsf{dequeue}(q)\\
\quad \left\{ v. \quad \begin{array}{l} (\ulcorner xs = \mathsf{nil}\urcorner * \ulcorner v = \mathsf{inj}_1\ ()\urcorner * \mathsf{isQueueSeq}(q, \mathsf{nil}, \Phi)) \ \lor \\ (\exists ts, tid, w, xs'.\ \ulcorner xs = (ts, tid, w) :: xs'\urcorner * \ulcorner v = \mathsf{inj}_2\ (ts, tid, w)\urcorner * \Phi(w) * \mathsf{isQueueSeq}(q, xs', \Phi)) \end{array} \right\}\\
\land\\
\forall q, \Phi, xs.\\
\qquad \{\mathsf{isQueueSeq}(q, xs, \Phi)\}\\
\qquad\quad \mathsf{readFront}(q)\\
\quad \left\{ v. \quad \begin{array}{l} \mathsf{isQueueSeq}(q, xs, \Phi) * \\ (\ulcorner xs = \mathsf{nil}\urcorner * \ulcorner v = \mathsf{inj}_1\ ()\urcorner \ \lor \ (\exists ts, tid, w, xs'.\ \ulcorner xs = (ts, tid, w) :: xs'\urcorner * \ulcorner v = \mathsf{inj}_2\ (ts, tid, w)\urcorner)) \end{array} \right\}.
\end{array}
$$

The predicate isQueueSeq is not duplicable, and it is thus required in the precondition of each operation and re-established in the postconditions. The predicate isQueueSeq mentions both the linked list representing the queue its associated mathematical list of queue elements. This means we can capture exactly how each operation changes the contents of the queue; for example, after an enqueue operation, the enqueued value is appended to the list, and after the dequeue operation on a non-empty queue, the first element of the list is removed and returned.
Furthermore, the specification guarantees that all queue elements satisfy a predicate Φ chosen by the client. The enqueuer must prove $\Phi(w)$ for any value w that it wants to enqueue, and correspondingly, the dequeuer is guaranteed $\Phi(w)$ for any dequeued value w.
The specification of readFront is the same for both the enqueuer and dequeuer variants. Its postcondition is similar to the postcondition of dequeue apart from two differences: The list of queue elements is unchanged after reading, and we do not get $\Phi(w)$ for the read value w. This is because $\Phi(w)$ is not necessarily duplicable.

6.2 isQueueSeq Predicate

$$
\begin{array}{rl}
\mathsf{isQueueSeq}(v, xs, \Phi) \triangleq & \exists \mathsf{Q}, \mathsf{First}, \mathsf{Last}, \mathsf{FreeLater}, \mathsf{Announce}, \mathsf{Help}, \ell_{\mathsf{First}}, \ell_{\mathsf{Last}}, \ell_{\mathsf{FL}}, \ell^?_{\mathsf{A}}, v^?_{\mathsf{Help}}, v^?_{\mathsf{FL}}.\\
& \ulcorner v = \#\mathsf{Q}\urcorner * \mathsf{Q} \mapsto_{\square} (\mathsf{First}, \mathsf{Last}, \mathsf{FreeLater}, \mathsf{Announce}, \mathsf{Help})\\
& \mathsf{First} \mapsto \ell_{\mathsf{First}} * \mathsf{isNodeList}(\ell_{\mathsf{First}}, \ell_{\mathsf{Last}}, xs) * \mathsf{forAll}(xs, \Phi) *\\
& \mathsf{Last} \mapsto \ell_{\mathsf{Last}} * \ell_{\mathsf{Last}} \mapsto \mathsf{inj}_1\ () *\\
& \mathsf{FreeLater} \mapsto \ell_{\mathsf{FL}} * \mathsf{isOptPts}(\ell_{\mathsf{FL}}, v^?_{\mathsf{FL}}) *\\
& \mathsf{isOptPt}(\mathsf{Announce}, \ell^?_{\mathsf{A}}) * \mathsf{isOptVal}(\mathsf{Help}, v^?_{\mathsf{Help}}).
\end{array}
$$

The queue predicate takes as argument the mathematical list xs of queue elements, and the client-chosen predicate Φ that all queue elements must satisfy.

The predicate existentially quantifies the shared queue variables First, Last, FreeLater, Announce, and Help, together with the locations and values they point to: $\ell_{\mathsf{First}}, \ell_{\mathsf{Last}}, \ell_{\mathsf{FL}}, \ell^?_{\mathsf{A}}$, and $v^?_{\mathsf{Help}}$.

We first define the predicate $\mathsf{forAll}(xs, \Phi)$, which states that all queue values satisfy the predicate Φ:

$$\mathsf{forAll}(xs, \Phi) \triangleq \ulcorner xs = \mathsf{nil} \urcorner \lor \exists ts, tid, w, xs'.\ \ulcorner xs = (ts, tid, w) :: xs' \urcorner * \Phi(w) * \mathsf{forAll}(xs', \Phi).$$

Next, we define the recursive predicate $\mathsf{isNodeList}(\ell_{\mathsf{First}}, \ell_{\mathsf{Last}}, xs)$, which relates the linked list representation of the queue to the mathematical list xs:

$$\begin{aligned}\mathsf{isNodeList}(\ell_{\mathsf{First}}, \ell_{\mathsf{Last}}, xs) \triangleq\ & \left(\ulcorner xs = \mathsf{nil} \urcorner * \ulcorner \ell_{\mathsf{First}} = \ell_{\mathsf{Last}} \urcorner\right) \\ & \lor \\ & \big(\exists ts, tid, w.\ \ulcorner xs = (ts, tid, w) :: xs' \urcorner * \\ & \quad \exists \mathsf{next}, \ell.\ \ell_{\mathsf{First}} \mapsto \mathsf{inj}_2\ ((ts, tid, w), \mathsf{next}) * \\ & \quad\quad \mathsf{next} \mapsto \ell * \mathsf{isNodeList}(\ell, \ell_{\mathsf{Last}}, xs')\big).\end{aligned}$$

In the case where the list of queue elements is empty, First and Last point to the same location ($\ell_{\mathsf{First}} = \ell_{\mathsf{Last}}$). Otherwise, ℓ_{First} points to a pair of the head of the list and location pointing to the next node, $\ell_{\mathsf{First}} \mapsto \mathsf{inj}_2\ ((ts, tid, w), \mathsf{next})$. The next location points to location ℓ which recursively satisfies the isNodeList predicate for the same last location (ℓ_{Last}) and the tail of the list. This extra layer of indirection – the intermediate location, next, – follows our HeapLang representation of the queue (fig. 5.2).

Finally, we define predicates for optional locations and values. These connect the option types used in the specification to the HeapLang constructions $\mathsf{inj}_1\ ()$ and $\mathsf{inj}_2\ (v)$:

$$\begin{aligned}\mathsf{isOptPts}(\ell_{\mathsf{FL}}, v^?_{\mathsf{FL}}) \triangleq\ & \left(\ulcorner v^?_{\mathsf{FL}} = \mathtt{None} \urcorner * \ell_{\mathsf{FL}} \mapsto \mathsf{inj}_1\ ()\right) \lor \\ & \big(\exists ts, tid, w, \mathsf{next}, \ell.\ \ulcorner v^?_{\mathsf{FL}} = \mathtt{Some}(ts, tid, w, \mathsf{next}, \ell) \urcorner * \\ & \quad \ell_{\mathsf{FL}} \mapsto \mathsf{inj}_2\ ((ts, tid, w), \mathsf{next}) * \mathsf{next} \mapsto \ell\big).\end{aligned}$$

$$\begin{aligned}\mathsf{isOptPt}(\mathsf{Announce}, \ell^?_{\mathsf{A}}) \triangleq\ & \left(\ulcorner \ell^?_{\mathsf{A}} = \mathtt{None} \urcorner * \mathsf{Announce} \mapsto \mathsf{inj}_1\ ()\right) \lor \\ & \big(\exists \ell.\ \ulcorner \ell^?_{\mathsf{A}} = \mathtt{Some}(\ell) \urcorner * \mathsf{Announce} \mapsto \mathsf{inj}_2\ (\ell)\big)\end{aligned}$$

$$\begin{aligned}\mathsf{isOptVal}(\mathsf{Help}, v^?_{\mathsf{Help}}) \triangleq\ & \left(\ulcorner v^?_{\mathsf{Help}} = \mathtt{None} \urcorner * \mathsf{Help} \mapsto \mathsf{inj}_1\ ()\right) \lor \\ & \big(\exists ts, tid, w.\ \ulcorner v^?_{\mathsf{Help}} = \mathtt{Some}(ts, tid, w) \urcorner * \mathsf{Help} \mapsto \mathsf{inj}_2\ (ts, tid, w)\big)\ .\end{aligned}$$

Note that in $\mathsf{isOptPts}(\ell_{\mathsf{FL}}, v^?_{\mathsf{FL}})$ we include the intermediate points-to assertion $\mathsf{next} \mapsto \ell$. This is necessary for the proof of the dequeue operation, where the node stored in FreeLater may be freed.

6.3 Proofs of the Sequential Specification

In this section, we prove the specifications of the four queue operations in the sequential setting. The purpose of this setting is first of all to prove partial correctness of the queue operations with respect to the sequential specification. Second of all, in this setting without concurrency, we get a good understanding of how the linked list representation of the queue and the shared variables are modified in each of the four queue operations.
When we later prove the concurrent specifications (sections 7 and 8), we will not go through the proofs with the same amount of details. Instead, we focus on the additional reasoning required for concurrency, such as the ghost state and invariants that enable sharing and communication between the enqueuer and the dequeuer.

6.3.1 Initialisation

We want to prove the following specification for initialising the queue:

$$\forall \Phi,\ \{\mathsf{True}\}\ \mathsf{init}()\ \{\mathsf{v}.\ \mathsf{isQueueSeq}(v, \mathsf{nil}, \Phi)\}$$

We go through the init program (5.3.1).

We allocate two empty nodes $\ell_{\mathsf{First}} \mapsto \mathsf{inj}_1\ ()$ and $\ell_{\mathsf{FL}} \mapsto \mathsf{inj}_1\ ()$, and then we allocate two pointers First and Last to the first empty node ℓ_{First}, $\mathsf{First} \mapsto \ell_{\mathsf{First}}$ and $\mathsf{Last} \mapsto \ell_{\mathsf{First}}$. Consequently, $\ell_{\mathsf{First}} = \ell_{\mathsf{Last}}$. We allocate a pointer FreeLater pointing to the other empty node ℓ_{FL}, $\mathsf{FreeLater} \mapsto \ell_{\mathsf{FL}}$.
We allocate the other shared variables, Announce and Help, as pointers to $\mathsf{inj}_1\ ()$. This illustrates the fact that these variables are initially `None`.
Finally, we allocate a tuple of all shared variables, which we consider to be the shared queue, Q. We make this points-to assertion persistent, such that both the dequeuer and enqueuer agree on the shared variables: $Q \mapsto_{\square} (\mathsf{First}, \mathsf{Last}, \mathsf{FreeLater}, \mathsf{Announce}, \mathsf{Help})$.

Since $\ell_{\mathsf{First}} = \ell_{\mathsf{Last}}$, we satisfy $\mathsf{isNodeList}(\ell_{\mathsf{First}}, \ell_{\mathsf{Last}}, xs)$, for $xs = \mathsf{nil}$. Furthermore, nil trivially satisfies $\mathsf{forAll}(\mathsf{nil}, \Phi)$.

With these points-to assertions and additional propositions, we can satisfy the isQueueSeq predicate with $v = Q$ and an empty list of queue elements.

6.3.2 Enqueue

$\forall q, \Phi, xs, ts, tid, w.$

$\{\mathsf{isQueueSeq}(q, xs, \Phi) * \Phi(w)\}\ \mathsf{enqueue}(q, ts, tid, w)\ \{v.\ulcorner v = ()\urcorner * \mathsf{isQueueSeq}(q, xs +\!\!+ [(ts, tid, w)])\}.$

We go through the enqueue program (5.3.2).

From the precondition, we have $\mathsf{isQueueSeq}(q, xs, \Phi)$ and $\Phi(w)$.
First, we allocate the new last node, $\ell_{\mathsf{newLast}} \mapsto \mathsf{inj}_1\ ()$. Next, we allocate a reference to the location ℓ_{newLast}, $\mathsf{next} \mapsto \ell_{\mathsf{newLast}}$.

We own $\mathsf{Last} \mapsto \ell_{\mathsf{Last}}$ from the isQueueSeq predicate for some location ℓ_{Last}. Furthermore, we own the points-to $\ell_{\mathsf{Last}} \mapsto \mathsf{inj}_1\ ()$.

We store the tuple $((ts, tid, w), \mathsf{next})$ in ℓ_{Last}, such that we now own $\ell_{\mathsf{Last}} \mapsto ((ts, tid, w), \mathsf{next})$. Finally, we store ℓ_{newLast} in Last, giving us the points-to $\mathsf{Last} \mapsto \ell_{\mathsf{newLast}}$.

We now have to show that the isQueueSeq is satisfied with an updated list of elements, i.e.

$$\mathsf{isQueueSeq}(q, xs +\!\!+ [(ts, tid, w)]), \Phi).$$

The existentially quantified variables in $\mathsf{isQueueSeq}(q, xs, \Phi)$ remain unchanged, except ℓ_{Last}, which we now set to $\ell_{newLast}$.
We have to prove

$$\mathsf{isNodeList}(\ell_{\mathsf{First}}, \ell_{\mathsf{newLast}}, xs +\!\!+ [(ts, tid, w)])$$

given $\mathsf{isNodeList}(\ell_{\mathsf{First}}, \ell_{\mathsf{Last}}, xs)$ and

$$\mathsf{forAll}(xs +\!\!+ [(ts, tid, w)], \Phi)$$

given $\mathsf{forAll}(xs, \Phi)$.
The former is provable by induction on the list xs, and the latter follows from our assumption $\Phi(w)$.

With this, we can satisfy the postcondition, $\mathsf{isQueueSeq}(q, xs +\!\!+ [(ts, tid, w)]), \Phi)$.

6.3.3 Dequeue

$\forall q, \Phi, xs.\ \{\mathsf{isQueueSeq}(q, xs, \Phi)\}\ \mathsf{dequeue}(q)$

$\{v.\ (\ulcorner xs = \mathsf{nil}\urcorner * \ulcorner v = \mathsf{inj}_1\ ()\urcorner * \mathsf{isQueueSeq}(q, \mathsf{nil}, \Phi))$

$\vee\ (\exists ts, tid, w.\ulcorner xs = (ts, tid, w) :: xs'\urcorner * \ulcorner v = \mathsf{inj}_2\ (ts, tid, w)\urcorner * \Phi(w) * \mathsf{isQueueSeq}(q, xs', \Phi))\ \}$

We go through the implementation of the dequeue operation (5.3.3).
We get the assumption $\mathsf{isQueueSeq}(q, xs, \Phi)$ from the precondition. Among other things, we own the points-to $\mathsf{First} \mapsto \ell_{\mathsf{First}}$. From this, we load the current first location, ℓ_{First}.

We own the points-to $\mathsf{Last} \mapsto \ell_{\mathsf{Last}}$, so we load the location pointing to the last element of the queue, ℓ_{Last}, and compare it to ℓ_{First}.
If $\ell_{\mathsf{First}} = \ell_{\mathsf{Last}}$, we conclude that the list of queue elements xs is empty from the assumptions:

$$\mathsf{isNodeList}(\ell_{\mathsf{First}}, \ell_{\mathsf{Last}}, xs) * \ell_{\mathsf{Last}} \mapsto \mathsf{inj}_1\ ().$$

We return $\mathsf{inj}_1\ ()$ and use $\mathsf{isQueueSeq}(q, xs, \Phi)$ unchanged to satisfy the LHS of the disjunction in the postcondition.
Otherwise, if $\ell_{\mathsf{First}} \neq \ell_{\mathsf{Last}}$, we conclude from the assumption $\mathsf{isNodeList}(\ell_{\mathsf{First}}, \ell_{\mathsf{Last}}, xs)$ that the list xs is not empty, i.e. $\exists ts, tid, w, xs'.\ xs = (ts, tid, w) :: xs'$.
By unfolding $\mathsf{isNodeList}(\ell_{\mathsf{First}}, \ell_{\mathsf{Last}}, (ts, tid, w) :: xs')$, we get two points-to assertions,

$$\ell_{\mathsf{First}} \mapsto \mathsf{inj}_2\ ((ts, tid, w), \mathsf{next}) * \mathsf{next} \mapsto \ell_{\mathsf{next}},$$

for some locations next and ℓ_{next}. Furthermore, we get $\mathsf{isNodeList}(\ell_{\mathsf{next}}, \ell_{\mathsf{Last}}, xs')$.
We load the pair $((ts, tid, w), \mathsf{next})$ stored in ℓ_{First}, store the queue element value $\mathsf{inj}_2\ (ts, tid, w)$ in Help, and then update the front element of the queue to be the next node by storing ℓ_{next} in First.

Next, we check whether the previous first element (ℓ_{First}) has been announced by comparing $\mathsf{inj}_2\ \ell_{\mathsf{First}}$ to the location stored in $\mathsf{Announce}$, $\ell_{\mathsf{A}}^?$:

If $\mathsf{inj}_2\ \ell_{\mathsf{First}} \neq \ell_{\mathsf{A}}^?$, then we simply free the points-tos $\ell_{\mathsf{First}} \mapsto \mathsf{inj}_2\ ((ts, tid, w), \mathsf{next})$ and $\mathsf{next} \mapsto \ell_{\mathsf{next}}$. We return the value (ts, tid, w).
To prove the postcondition, it suffices to show

$$\Phi(w) * \mathsf{isQueueSeq}(q, xs', \Phi).$$

We re-establish the queue predicate using ℓ_{next} as the new first location.
We can prove $\mathsf{isNodeList}(\mathsf{next}, \ell_{\mathsf{Last}}, xs')$ using $\mathsf{isNodeList}(\ell_{\mathsf{next}}, \ell_{\mathsf{Last}}, xs')$ and the points-to assertion $\mathsf{next} \mapsto \ell_{\mathsf{next}}$.
Finally, we split the predicate $\mathsf{forAll}(xs, \Phi)$ into a predicate for the head and the tail, $\mathsf{forAll}((ts, tid, w), \Phi)$ and $\mathsf{forAll}(xs', \Phi)$.
Finally, we set $v_{\mathsf{Help}}^?$ to $\mathsf{inj}_2\ (ts, tid, w)$ and obtain the required postcondition.

Otherwise, if $\mathsf{inj}_2\ \ell_{\mathsf{First}} = \ell_{\mathsf{A}}^?$, it is not safe to free the location ℓ_{First}. Instead, we read the current location stored in $\mathsf{FreeLater}$, ℓ_{FL}, and store ℓ_{First} in $\mathsf{FreeLater}$.
We own $\mathsf{isOptPts}(\ell_{\mathsf{FL}}, v_{\mathsf{FL}}^?)$. There are two cases:
Either $\ell_{\mathsf{FL}} \mapsto \mathsf{inj}_1\ ()$, in which case we just free $\ell_{\mathsf{FL}} \mapsto \mathsf{inj}_1\ ()$.
Otherwise, $\ell_{\mathsf{FL}} \mapsto \mathsf{inj}_2\ ((ts, tid, w), \mathsf{next}_{\mathsf{FL}})$ and $\mathsf{next}_{\mathsf{FL}} \mapsto \ell_{\mathsf{FL}}$. In this case, we free both points-tos.
After this, we satisfy the postcondition in a similar way as in the case where $\mathsf{inj}_2\ \ell_{\mathsf{First}} \neq \ell_{\mathsf{A}}^?$. The only difference is that the previous first node is now stored in $\mathsf{FreeLater}$, so we set ℓ_{FL} to be ℓ_{First} and $v_{\mathsf{FL}}^?$ to be $\mathsf{Some}(ts, tid, w, \mathsf{next}, \ell_{\mathsf{next}})$.

6.3.4 readFront (Enqueuer)

$$\forall q, \Phi, xs.\ \{\mathsf{isQueueSeq}(q, xs, \Phi)\}\ \mathsf{readFront}(q)\ \Big\{v.\ \mathsf{isQueueSeq}(q, xs, \Phi) * \big(\ulcorner xs = \mathsf{nil} \urcorner * \ulcorner v = \mathsf{inj}_1\ () \urcorner \lor \big(\exists ts, tid, w.\ \ulcorner xs = (ts, tid, w) :: xs' \urcorner * \ulcorner v = \mathsf{inj}_2\ (ts, tid, w) \urcorner\big)\big)\Big\}.$$

We go through the implementation of the $\mathsf{readFront}$ operation for the enqueuer (5.3.4).

We get $\mathsf{isQueueSeq}(q, xs, \Phi)$ from the precondition. We read the location ℓ_{First} stored in First, and the location ℓ_{Last} stored in Last.
If $\ell_{\mathsf{First}} = \ell_{\mathsf{Last}}$, we can conclude that the list of queue elements xs is empty, and we return the value $\mathsf{inj}_1\ ()$, and we satisfy the postcondition by proving $\mathsf{isQueueSeq}(q, xs, \Phi)$ without any changes.

If $\ell_{\mathsf{First}} \neq \ell_{\mathsf{Last}}$, we can conclude that the list of queue elements xs is non-empty, i.e. $\exists ts, tid, w, xs'.\ xs = (ts, tid, w) :: xs'$. Furthermore, by unfolding $\mathsf{isNodeList}(\ell_{\mathsf{First}}, \ell_{\mathsf{Last}}, (ts, tid, w) :: xs')$, we get a points-to assertion, $\ell_{\mathsf{First}} \mapsto \mathsf{inj}_2\ ((ts, tid, w), \mathsf{next})$ for some location next.

We announce the location ℓ_{First} by storing $\mathsf{inj}_2\,(\ell_{\mathsf{First}})$ in $\mathsf{Announce}$. Then, we check whether the location ℓ_{First} that First points to has changed since the first time we loaded it. Since we are in a sequential setting, we own the points-to assertion $\mathsf{First} \mapsto \ell_{\mathsf{First}}$. This means we are always in the case that the first location remains unchanged, $\ell_{\mathsf{First}} = \ell_{\mathsf{First}}$. Therefore, we return the value $\mathtt{Some}(ts, tid, w)$, and we provide $\mathsf{isQueueSeq}(q, xs, \Phi)$ unchanged, except for the updated announced location, $\mathtt{Some}(\ell_{\mathsf{First}})$.

6.3.5 readFront (Dequeuer)

The proof of readFront for the dequeuer is exactly the same as for the enqueuer in the sequential setting, except we do not announce the location of the first node ℓ_{First}.

7 Concurrent Specifications & Proofs (Without Freeing)

In this section, we present and prove a concurrent specification for the SESD queue algorithm. In this setting, we consider two threads operating on the linked list representing the queue concurrently: an enqueuer and a dequeuer.
We first study a simplified version of the algorithm in which the dequeuer does not perform memory freeing in the dequeue operation. This simplification allows us to focus on the concurrent interactions between the enqueuer and dequeuer. We will verify the full SESD algorithm including freeing in section 8. The main focus of this section is the interactions between the two threads and defining a useful duplicable queue predicate with an invariant.

We introduce four differences from the sequential version:

1. The isQueueConc predicate is **persistent**. This is necessary, because it must be shared between the enqueuer and the dequeuer. We make it persistent by using an **invariant** containing half points-tos of the shared variables, and we give the other half to either the dequeuer or the enqueuer, depending on the variable. Additionally, we make the points-tos pointing to nodes in the linked list representing the queue persistent. However, by making the points-tos persistent, we cannot free them. We explain this in section 7.1.

2. The invariant takes into account the **intermediate step in enqueue** where the location ℓ_{Last} that the global pointer Last points to, does *not* point to the empty dummy node, but instead to the newly enqueued node. We handle this by introducing an auxiliary location, ℓ_{dummy}, that *always* points to the dummy node (the empty node at the end of the queue). We will go through this step in section 7.4.

3. We use a notion of **reachability and abstract reachability** in the linked list representing the queue, introduced in [VB21] and used in [Ped24]. Reachability between the nodes in the queue is necessary in the dequeue operation, where we must ensure that the new first location satisfies the isNodeList predicate for the tail of the list. In particular, we want to know that the first node of the linked list reaches the last node. This is defined and explained in section 7.5.

4. We use the **one-shot resource algebra** to rule out an impossible case where the value stored in the global variable Help is `None` in the readFrontEnq operation. The optional $v^?_{\mathsf{Help}}$ value stored in Help is initially `None`, but as soon as the dequeuer has performed the dequeue operation from a non-empty queue, the $v^?_{\mathsf{Help}}$ value remains `Some`. We will go through the proof of readFrontEnq in section 7.6, focussing on reading the Help value.

In the following sections, we will explain these differences from the sequential version in more detail, including motivating and explaining how they are used in the proofs of the specification of each operation.

We use coloured highlights to emphasise explanations of and changes in definitions.

* Yellow highlights are used to connect an explanation of a definition to its context, for example when explaining individual parts of a big invariant.

* Blue highlights indicate additions or differences between similar definitions.

7.1 A persistent isQueueConc Predicate

The isQueueConc predicate must be persistent to share it between both the enqueuer and the dequeuer. Therefore, we define the isQueueConc predicate using a persistent points-to pointing to the shared variables, and an invariant for the non-persistent parts of the queue.

$$\begin{aligned}\mathsf{isQueueConc}(v, G, \Phi) \triangleq \exists \mathsf{Q}, \mathsf{First}, \mathsf{Last}, \mathsf{FreeLater}, \mathsf{Announce}, \mathsf{Help}, \iota.\\ \ulcorner v = \#\mathsf{Q} \urcorner * \mathsf{Q} \mapsto_{\square} (\mathsf{First}, \mathsf{Last}, \mathsf{FreeLater}, \mathsf{Announce}, \mathsf{Help})\\ \boxed{\mathsf{queueInv}(\mathsf{First}, \mathsf{Last}, \mathsf{Announce}, \mathsf{Help}, G, \Phi)}^{\iota}.\end{aligned}$$

7.1.1 Queue Invariant

We define the invariant to give an overview, and we highlight the differences from the sequential queue predicate (section 6.2).
First, we define the ghost names used in the invariant.

$$G \triangleq (\ \gamma_{\mathsf{First}}, \gamma_{\mathsf{Last}}, \gamma_{\mathsf{Dummy}}, \quad \text{(Abstract Reachability)}$$
$$\gamma_{\mathsf{help}}, \gamma_{\mathsf{init}}\ ). \quad \text{(One-Shot RA for Help)}$$

$$\begin{aligned}\mathsf{queueInv}(\mathsf{First}, \mathsf{Last}, \mathsf{Announce}, \mathsf{Help}, G, \Phi) \triangleq\ & \exists \ell_{\mathsf{First}}, \ell_{\mathsf{Last}}, xs, \ell_{\mathsf{A}}^{?}, v_{\mathsf{Help}}^{?},\ \ell_{\mathsf{dummy}}, \ell_{\mathsf{init}}\ .\\ & \mathsf{First} \mapsto_{1/2} \ell_{\mathsf{First}} * \mathsf{Last} \mapsto_{1/2} \ell_{\mathsf{Last}} *\\ & \mathsf{isNodeList}(\ell_{\mathsf{First}}, \ell_{\mathsf{dummy}}, xs) * \mathsf{forAll}(xs, \Phi) *\\ & \ell_{\mathsf{dummy}} \mapsto_{1/2} \mathsf{inj}_1\ () * \mathsf{lastDisj}(\ell_{\mathsf{Last}}, \ell_{\mathsf{dummy}}) *\\ & \mathsf{isOptPt}(\mathsf{Announce}, \ell_{\mathsf{A}}^{?}) * \mathsf{isOptVal}(\mathsf{Help}, v_{\mathsf{Help}}^{?}) *\\ & \mathsf{absReach}(\gamma_{\mathsf{First}}, \gamma_{\mathsf{Last}}, \gamma_{\mathsf{Dummy}}, \ell_{\mathsf{First}}, \ell_{\mathsf{Last}}, \ell_{\mathsf{dummy}}) *\\ & \mathsf{helpValOneShot}(\gamma_{\mathsf{help}}, \gamma_{\mathsf{init}}, \ell_{\mathsf{First}}, \ell_{\mathsf{init}}, v_{\mathsf{Help}}^{?})\ .\end{aligned}$$

We will go through the individual parts of the invariant in the following sections, justifying their necessity for proving the specifications.

7.2 Distributing the Points-tos and Shared Variables

One of the main changes compared to the sequential queue predicate is that we now store points-tos in the invariant. In the sequential setting, the points-tos for all global variables – First, Last, FreeLater, Announce and Help – were passed between the enqueuer and dequeuer in a sequential manner. In this concurrent version, where we have a single dequeuer and a single enqueuer, we will distribute (parts of) the points-tos between the dequeuer, the enqueuer and the invariant.

First, we consider which points-tos the dequeuer and enqueuer read (R) and/or write (W) to during the operations:

Variable	Dequeuer	Enqueuer
First	R,W	R
Last	R	R,W
Announce	R	W
Help	W	R
FreeLater	R,W	-

Table 1: Dequeuer and enqueuer interactions with shared variables.

We give half a points-to to a thread, say the dequeuer, if it writes to the location, and we keep the other half in the invariant. This way, the dequeuer can combine its half with the half in the invariant when storing into the location, and the enqueuer can read from location using the part of the points-to in the invariant.
Since only the dequeuer writes and reads from FreeLater, it gets the entire points-to.

Furthermore, we consider how the enqueuer and dequeuer interact with the locations pointed to by First, Last and FreeLater – i.e. $\ell_{\mathsf{First}}, \ell_{\mathsf{Last}}$ and ℓ_{FL}.

Variable	Dequeuer	Enqueuer
ℓ_{First}	R	-
ℓ_{Last}	R	R,W
ℓ_{FL}	R,W	-

Table 2: Dequeuer and enqueuer interactions with other locations.

The location ℓ_{First} is either a persistent points-to pointing to a non-empty node, or, if the queue is empty, $\ell_{\mathsf{First}} = \ell_{\mathsf{Last}}$ and $\ell_{\mathsf{Last}} \mapsto \mathsf{inj}_1\ ()$. Since the enqueuer writes to ℓ_{Last}, we give $\ell_{\mathsf{Last}} \mapsto_{1/2} \mathsf{inj}_1\ ()$ to the enqueuer.
Similarly, ℓ_{FL} is either a persistent points-to pointing to a node in the linked list representing the queue, or it points to a dummy node separate from the queue. Since only the dequeuer reads from and writes to this location, we give $\ell_{\mathsf{FL}} \mapsto v^?_{\mathsf{FL}}$ to the dequeuer.

To conclude, we distribute the shared variables between the three "parties" as follows:

- Dequeuer: $\mathsf{First} \mapsto_{1/2} \ell_{\mathsf{First}} * \mathsf{FreeLater} \mapsto \ell_{\mathsf{FL}} * \mathsf{isOptVal}(\mathsf{Help}, v^?_{\mathsf{Help}}) * \mathsf{isOptPts}(\ell_{\mathsf{FL}}, v^?_{\mathsf{FL}})$.
- Enqueuer: $\mathsf{Last} \mapsto_{1/2} \ell_{\mathsf{Last}} * \mathsf{isOptPt}(\mathsf{Announce}, \ell^?_{\mathsf{A}}) * \ell_{\mathsf{Last}} \mapsto_{1/2} \mathsf{inj}_1\ ()$.
- Invariant:

$$\begin{aligned}&\mathsf{First} \mapsto_{1/2} \ell_{\mathsf{First}} * \mathsf{Last} \mapsto_{1/2} \ell_{\mathsf{Last}} * \ell_{\mathsf{Last}} \mapsto_{1/2} \mathsf{inj}_1\ ()\\ &* \mathsf{isOptPt}(\mathsf{Announce}, \ell^?_{\mathsf{A}}) * \mathsf{isOptVal}(\mathsf{Help}, v^?_{\mathsf{Help}}).\end{aligned}$$

7.2.1 Persistent isNodeList Predicate and Shared Variables

We use a *persistent* predicate for the linked list representing the queue. The new isNodeList predicate arises from the previous isNodeList predicate (see section 6.2) by exchanging the normal points-tos for persistent points-tos. Furthermore, in the invariant, we set the end of the list to be a dummy node ℓ_{dummy}, which always points to the empty dummy node. A persistent isNodeList predicate allows the enqueuer and dequeuer to keep a view of the queue simultaneously.
Intuitively, the linked list representing the queue is persistent in the sense that once a node is added, its value and its place in the list (i.e. its next pointer) never change. The only exception is the dummy node, $\ell_{\mathsf{Last}} \mapsto \mathsf{inj}_1\ ()$, which cannot be persistent, since it is updated when the enqueuer enqueues a new value into the queue.

However, it is not sound to free persistent points-to assertions. The intuition for why the algorithm is still sound is that the points-tos are persistent only between the time when a value has been enqueued into them, and until the value is dequeued. In this section, we do not yet reason about memory freeing. We consider the full SESD algorithm including memory freeing in section 8.

We revisit $\mathsf{isOptPts}(\ell_{\mathsf{FL}}, v^?_{\mathsf{FL}})$, $\mathsf{isOptPt}(\mathsf{Announce}, \ell^?_{\mathsf{A}})$ and $\mathsf{isOptVal}(\mathsf{Help}, v^?_{\mathsf{Help}})$ from isQueueSeq (see section 6.2).

The isOptPts predicate now uses a *persistent points-to* for the non-empty node case:

$$\begin{aligned}\mathsf{isOptPts}(\ell_{\mathsf{FL}}, v^?_{\mathsf{FL}}) \triangleq\ &\left(\ulcorner v^?_{\mathsf{FL}} = \texttt{None}\urcorner * \ell_{\mathsf{FL}} \mapsto \mathsf{inj}_1\ ()\right) \vee\\ &\Big(\exists ts, tid, w, \mathsf{next}, \ell.\ \ulcorner v^?_{\mathsf{FL}} = \texttt{Some}(ts, tid, w, \mathsf{next}, \ell)\urcorner *\\ &\ell_{\mathsf{FL}} \mapsto_{\square} \mathsf{inj}_2\ ((ts, tid, w), \mathsf{next}) * \mathsf{next} \mapsto_{\square} \ell\Big).\end{aligned}$$

Similarly, $\mathsf{isOptPt}(\mathsf{Announce}, \ell^?_{\mathsf{A}})$ and $\mathsf{isOptVal}(\mathsf{Help}, v^?_{\mathsf{Help}})$ now use *half points-tos* instead of full points-tos. This enables sharing the points-tos between the invariant and enqueuer and between the invariant and dequeuer respectively.

7.2.2 Queue Invariant

In this section, we have accounted for the distribution of the points-tos, and the changes to the isNodeList, isOptPts, isOptPt and isOptVal predicates. We highlight these in the invariant:

$$\begin{aligned}\mathsf{queueInv}(\mathsf{First}, \mathsf{Last}, \mathsf{Announce}, \mathsf{Help}, G, \Phi) \triangleq\ & \exists \ell_{\mathsf{First}}, \ell_{\mathsf{Last}}, xs, \ell_{\mathsf{A}}^?, v_{\mathsf{Help}}^?, \ell_{\mathsf{dummy}}, \ell_{\mathsf{init}}.\\ & \mathsf{First} \mapsto_{1/2} \ell_{\mathsf{First}} * \mathsf{Last} \mapsto_{1/2} \ell_{\mathsf{Last}} *\\ & \mathsf{isNodeList}(\ell_{\mathsf{First}}, \ell_{\mathsf{dummy}}, xs) * \mathsf{forAll}(xs, \Phi) *\\ & \ell_{\mathsf{dummy}} \mapsto_{1/2} \mathsf{inj}_1\ () * \mathsf{lastDisj}(\ell_{\mathsf{Last}}, \ell_{\mathsf{dummy}}) *\\ & \mathsf{isOptPt}(\mathsf{Announce}, \ell_{\mathsf{A}}^?) * \mathsf{isOptVal}(\mathsf{Help}, v_{\mathsf{Help}}^?) *\\ & \mathsf{absReach}(G, \ell_{\mathsf{First}}, \ell_{\mathsf{Last}}, \ell_{\mathsf{dummy}}) *\\ & \mathsf{helpValOneShot}(G, \ell_{\mathsf{First}}, \ell_{\mathsf{init}}, v_{\mathsf{Help}}^?).\end{aligned}$$

7.3 Concurrent Specification

We distribute parts of the points-to assertions and ghost resources between the invariant, the enqueuer, and the dequeuer. These resources are provided in the preconditions and re-established in the postconditions. To do this, we define two auxiliary predicates, isEnqueuer and isDequeuer, that combine all the resources that are given to each thread:

$$\begin{aligned}\mathsf{isEnqueuer}(G, \mathsf{Last}, \mathsf{Announce}) \triangleq\ & \exists \ell_{\mathsf{Last}}, \ell_{\mathsf{A}}^?.\\ & \mathsf{Last} \mapsto_{1/2} \ell_{\mathsf{Last}} *\\ & \mathsf{isOptPt}(\mathsf{Announce}, \ell_{\mathsf{A}}^?) *\\ & \ell_{\mathsf{Last}} \mapsto_{1/2} \mathsf{inj}_1\ ().\end{aligned}$$

$$\begin{aligned}\mathsf{isDequeuer}(G, \mathsf{First}, \mathsf{FreeLater}, \mathsf{Help}) \triangleq\ & \exists \ell_{\mathsf{First}}, \ell_{\mathsf{FL}}, v_{\mathsf{Help}}^?, v_{\mathsf{FL}}^?.\\ & \mathsf{First} \mapsto_{1/2} \ell_{\mathsf{First}} *\\ & \mathsf{FreeLater} \mapsto \ell_{\mathsf{FL}} *\\ & \mathsf{isOptVal}(\mathsf{Help}, v_{\mathsf{Help}}^?) *\\ & \mathsf{isOptPts}(\ell_{\mathsf{FL}}, v_{\mathsf{FL}}^?).\end{aligned}$$

The postcondition of init is the predicate $\mathsf{isQueueConc}(v, G, \Phi)$ together with the predicates for the enqueuer and dequeuer. Initially, all existentially quantified option values are `None`.
We pass around the persistent points-to assertion

$$Q \mapsto_{\square} (\mathsf{First}, \mathsf{Last}, \mathsf{FreeLater}, \mathsf{Announce}, \mathsf{Help})$$

to ensure agreement on the shared queue variables.
The specifications for enqueue and dequeue are weak. For example, always returning `None` as the result of dequeue and readFront would still satisfy the specification. Additionally, the specification does not specify FIFO queue behaviour. In fact, the specifications given here matches the specification for the concurrent bag in the Iris lecture notes ([BB23] Example 8.40, p. 74). We discuss how we could give stronger specifications in future work (section 9).
However, the client-chosen predicate Φ guarantees that any dequeued value was previously enqueued. The readFront operations do not include Φ, since $\Phi(w)$ is not necessarily duplicable. One could instead use a persistent predicate Ψ such that $\Phi(w) \vdash \square\, \Psi(w)$, which we also leave for future work.

$$
\begin{array}{l}
\exists\, \mathsf{isQueueConc} : \mathit{Val} \to \mathsf{GhostNames} \to (\mathit{Val} \to \mathit{iProp}) \to \mathit{iProp}, \\
\quad \forall v, G, \Phi.\ \mathsf{isQueueConc}(v, G, \Phi) \implies \Box\, \mathsf{isQueueConc}(v, G, \Phi) \\
\quad \wedge \\
\quad \forall \Phi.\ \{\mathsf{True}\} \\
\qquad\quad \mathsf{init}() \\
\qquad\quad \left\{ v.\ \begin{array}{l} \exists G, Q, \mathsf{First}, \mathsf{Last}, \mathsf{FreeLater}, \mathsf{Announce}, \mathsf{Help}. \\ \ulcorner v = Q \urcorner * \\ Q \mapsto_{\Box} (\mathsf{First}, \mathsf{Last}, \mathsf{FreeLater}, \mathsf{Announce}, \mathsf{Help}) * \\ \mathsf{isQueueConc}(v, G, \Phi) * \\ \mathsf{isEnqueuer}(G, \mathsf{Last}, \mathsf{Announce}) * \\ \mathsf{isDequeuer}(G, \mathsf{First}, \mathsf{FreeLater}, \mathsf{Help}) \end{array} \right\} \\
\quad \wedge \\
\quad \forall ts, tid, w, Q, G, \Phi, \mathsf{First}, \mathsf{Last}, \mathsf{FreeLater}, \mathsf{Announce}, \mathsf{Help}. \\
\qquad \left\{ \begin{array}{l} Q \mapsto_{\Box} (\mathsf{First}, \mathsf{Last}, \mathsf{FreeLater}, \mathsf{Announce}, \mathsf{Help}) * \\ \mathsf{isQueueConc}(Q, G, \Phi) * \\ \mathsf{isEnqueuer}(G, \mathsf{Last}, \mathsf{Announce}) * \\ \Phi(w) \end{array} \right\} \\
\qquad\quad \mathsf{enqueue}((ts, tid, w), Q) \\
\qquad \left\{ v.\ \begin{array}{l} \ulcorner v = () \urcorner * \\ \mathsf{isEnqueuer}(G, \mathsf{Last}, \mathsf{Announce}) \end{array} \right\} \\
\quad \wedge \\
\quad \forall Q, G, \Phi, \mathsf{First}, \mathsf{Last}, \mathsf{FreeLater}, \mathsf{Announce}, \mathsf{Help}. \\
\qquad \left\{ \begin{array}{l} Q \mapsto_{\Box} (\mathsf{First}, \mathsf{Last}, \mathsf{FreeLater}, \mathsf{Announce}, \mathsf{Help}) * \\ \mathsf{isQueueConc}(Q, G, \Phi) * \\ \mathsf{isDequeuer}(G, \mathsf{First}, \mathsf{FreeLater}, \mathsf{Help}) \end{array} \right\} \\
\qquad\quad \mathsf{dequeue}(Q) \\
\qquad \left\{ v.\ \begin{array}{l} (\ulcorner v = \mathsf{inj}_1\ () \urcorner) \vee (\exists ts, tid, w.\ \ulcorner v = \mathsf{inj}_2\ (ts, tid, w) \urcorner * \Phi(w)) \\ * \mathsf{isDequeuer}(G, \mathsf{First}, \mathsf{FreeLater}, \mathsf{Help}) \end{array} \right\} \\
\quad \wedge \\
\quad \forall Q, G, \Phi, \mathsf{First}, \mathsf{Last}, \mathsf{FreeLater}, \mathsf{Announce}, \mathsf{Help}. \\
\qquad \left\{ \begin{array}{l} Q \mapsto_{\Box} (\mathsf{First}, \mathsf{Last}, \mathsf{FreeLater}, \mathsf{Announce}, \mathsf{Help}) * \\ \mathsf{isQueueConc}(Q, G, \Phi) * \\ \mathsf{isEnqueuer}(G, \mathsf{Last}, \mathsf{Announce}) \end{array} \right\} \\
\qquad\quad \mathsf{readFrontEnq}(Q) \\
\qquad \left\{ v.\ \begin{array}{l} (\ulcorner v = \mathsf{inj}_1\ () \urcorner) \vee (\exists ts, tid, w.\ \ulcorner v = \mathsf{inj}_2\ (ts, tid, w) \urcorner) \\ * \mathsf{isEnqueuer}(G, \mathsf{Last}, \mathsf{Announce}) \end{array} \right\} \\
\quad \wedge \\
\quad \forall Q, G, \Phi, \mathsf{First}, \mathsf{Last}, \mathsf{FreeLater}, \mathsf{Announce}, \mathsf{Help}. \\
\qquad \left\{ \begin{array}{l} Q \mapsto_{\Box} (\mathsf{First}, \mathsf{Last}, \mathsf{FreeLater}, \mathsf{Announce}, \mathsf{Help}) * \\ \mathsf{isQueueConc}(Q, G, \Phi) * \\ \mathsf{isDequeuer}(G, \mathsf{First}, \mathsf{FreeLater}, \mathsf{Help}) \end{array} \right\} \\
\qquad\quad \mathsf{readFrontDeq}(Q) \\
\qquad \left\{ v.\ \begin{array}{l} (\ulcorner v = \mathsf{inj}_1\ () \urcorner) \vee (\exists ts, tid, w.\ \ulcorner v = \mathsf{inj}_2\ (ts, tid, w) \urcorner) \\ * \mathsf{isDequeuer}(G, \mathsf{First}, \mathsf{FreeLater}, \mathsf{Help}) \end{array} \right\}
\end{array}
$$

7.4 The enqueue operation

In the enqueue operation, depicted below in figure 7, the global pointer Last does not always point to the dummy node. In step 3 of the figure (or between lines 2-4 in the enqueue program, section 2.1.1), it points

to the newly enqueued node. The invariant takes this intermediate step into account by introducing an auxiliary location, ℓ_{dummy}, that *always* points to the queue's dummy node.

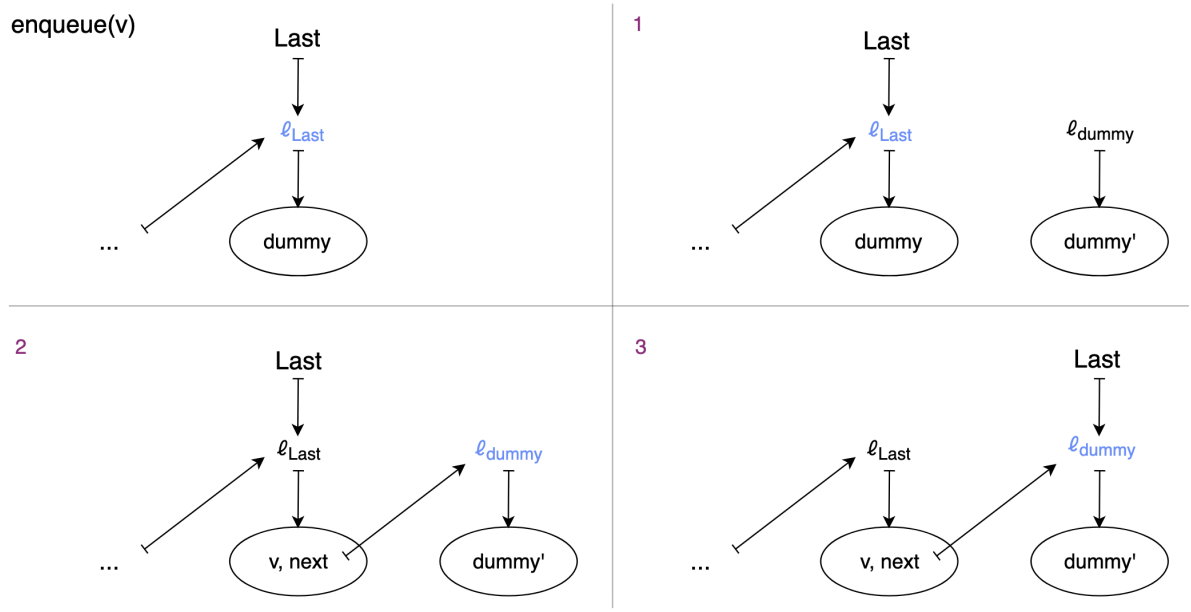

Figure 7: The enqueue operation. At each step, the current ℓ_{dummy} location is coloured blue.

We conclude that there are two possible cases for ℓ_{Last}.

1. The usual state where ℓ_{Last} points to the dummy node, i.e. $\ell_{\mathsf{Last}} \mapsto \mathsf{inj}_1\ ()$.
2. The intermediate step in the enqueue operation where ℓ_{Last} points to the node containing the newly enqueued value, i.e. $\ell_{\mathsf{Last}} \mapsto_\square (v, \mathsf{next}) * \mathsf{next} \mapsto \ell_{\mathsf{dummy}} * \ell_{\mathsf{dummy}} \mapsto \mathsf{inj}_1\ ()$.

The enqueuer should intuitively be able to know which case we are in, since only the enqueuer can switch between the cases. The dequeuer cannot know which case we are in, so it should be able to dequeue and read the front element regardless.
We encode the two cases as a disjunction.

$$\mathsf{lastDisj}(\ell_{\mathsf{Last}}, \ell_{\mathsf{dummy}}) \triangleq \ulcorner \ell_{\mathsf{Last}} = \ell_{\mathsf{dummy}} \urcorner \lor \left(\exists ts, tid, w, \mathsf{next}.\ \ell_{\mathsf{Last}} \mapsto_\square \mathsf{inj}_2\ ((ts, tid, w), \mathsf{next}) * \mathsf{next} \mapsto_\square \ell_{\mathsf{dummy}}\right).$$

7.4.1 Enqueue

The enqueuer wants to enqueue a value $v = (ts, tid, w)$ into the queue. We go through the program 5.3.2.

The enqueuer owns – among other things – the points-tos $\mathsf{Last} \mapsto_{1/2} \ell_{\mathsf{Last}}$ and $\ell_{\mathsf{Last}} \mapsto \mathsf{inj}_1\ ()$, and the $\mathsf{isQueueConc}(v, G, \Phi)$ predicate.

The enqueuer first allocates a new dummy node, $\ell'_{\mathsf{dummy}} \mapsto \mathsf{inj}_1\ ()$. Then, the enqueuer allocates a location, next, pointing to the new dummy location and makes it persistent: $\mathsf{next} \mapsto_\square \ell'_{\mathsf{dummy}}$.
Next, the enqueuer must store the the node with the new value $\mathsf{inj}_2\ ((ts, tid, w), \mathsf{next})$ in ℓ_{Last}. The enqueuer owns half of the points-to, $\ell_{\mathsf{Last}} \mapsto_{1/2} \mathsf{inj}_1\ ()$. To store a value in this location, we must get the other half of the points-to assertion from the invariant. We open the invariant.
From the disjunction $\mathsf{lastDisj}(\ell_{\mathsf{Last}}, \ell_{\mathsf{dummy}})$ in the invariant, we have two cases:

1. We are in the LHS of the disjunction, $\ulcorner \ell_{\mathsf{Last}} = \ell_{\mathsf{dummy}} \urcorner$. We combine the points-to $\ell_{\mathsf{Last}} \mapsto_{1/2} \mathsf{inj}_1\ ()$ from the enqueuer and $\ell_{\mathsf{dummy}} \mapsto_{1/2} \mathsf{inj}_1\ ()$ from the invariant. This gives us the full points-to for $\ell_{\mathsf{Last}} = \ell_{\mathsf{dummy}}$. We store the new node $\mathsf{inj}_2\ ((ts, tid, w), \mathsf{next})$ in ℓ_{Last} and make the points-to persistent. We close the invariant: The dummy location is now ℓ'_{dummy}, and we give half of the the points-to $\ell'_{\mathsf{dummy}} \mapsto_{1/2} \mathsf{inj}_1\ ()$. We close the ℓ_{Last}-disjunction in the RHS, since $\ell_{\mathsf{Last}} \mapsto_\square \mathsf{inj}_2\ ((ts, tid, w), \mathsf{next}) * \mathsf{next} \mapsto_\square \ell'_{\mathsf{dummy}}$.

2. We are in the RHS of the disjunction, so $\exists ts, tid, w, \mathsf{next}.\ \ell_{\mathsf{Last}} \mapsto_{\square} \mathsf{inj}_2\ ((ts, tid, w), \mathsf{next}) * \mathsf{next} \mapsto_{\square} \ell_{\mathsf{dummy}}$. This is a contradiction, since the enqueuer already owns $\ell_{\mathsf{Last}} \mapsto_{1/2} \mathsf{inj}_1\ ()$, and

$$\ell_{\mathsf{Last}} \mapsto_{\square} ((ts, tid, w), \mathsf{next}) * \ell_{\mathsf{Last}} \mapsto_{1/2} \mathsf{inj}_1\ () \vdash \mathsf{False}.$$

Finally, we want to store the new dummy location ℓ'_{dummy} in Last. We open the invariant again, and we combine the enqueuer's half of the points-to $\mathsf{Last} \mapsto_{1/2} \ell'_{\mathsf{dummy}}$ with the other half from the invariant to obtain the full points-to $\mathsf{Last} \mapsto \ell'_{\mathsf{dummy}}$.
After storing, we split $\mathsf{Last} \mapsto \ell'_{\mathsf{dummy}}$ into half for the enqueuer and half for the invariant. We close the invariant by letting the last location be ℓ'_{dummy}. We thus chose the LHS of the disjunction $\mathsf{lastDisj}(\ell'_{\mathsf{dummy}}, \ell'_{\mathsf{dummy}})$, where the last and the dummy locations are equal.

We return the unit value, and use $\mathsf{Last} \mapsto_{1/2} \ell'_{\mathsf{dummy}}$ and $\ell'_{\mathsf{dummy}} \mapsto_{1/2} \mathsf{inj}_1\ ()$ to satisfy the postcondition.

7.5 Reachability and Abstract Reachability

This notion of reachability and abstract reachability was introduced in connection with verifying the Michael Scott queue [MS96] in the paper [VB21]. The Michael Scott queue is modelled similarly to this queue, i.e. as a linked list with a dummy node, and with pointers to the first and last elements of the queue. However, in our version, Last points to an empty dummy node, while in the Michael-Scott representation of the queue, it points to the last *non-empty* node.

Definition 7.1. (Reachability)
The reachability relation is defined as a least fixed point:

$$\begin{aligned}\ell_n \rightsquigarrow \ell_m \triangleq\ & \ulcorner \ell_n = \ell_m \urcorner \vee \\ & \exists ts, tid, w, \mathsf{next}, \ell_o.\ \ell_n \mapsto_{\square} \mathsf{inj}_2\ ((ts, tid, w), \mathsf{next}) * \mathsf{next} \mapsto_{\square} \ell_o * \ell_o \rightsquigarrow \ell_m.\end{aligned}$$

This definition differs slightly from the original definition in [VB21] in that it is reflexive for any location, even those pointing to an empty node ($\mathsf{inj}_1\ ()$).
Furthermore, we note that the reachability is a relation only between locations pointing to nodes, *not* including the intermediate locations. For example, consider location ℓ_n satisfying $\ell_n \mapsto_{\square} \mathsf{inj}_2\ (v, \mathsf{next}) * \mathsf{next} \mapsto_{\square} \ell_o$, for some value v and locations next and ℓ_o. Here, $\ell_n \rightsquigarrow \ell_o$, but it is not the case that $\ell_n \rightsquigarrow \mathsf{next}$ or that $\mathsf{next} \rightsquigarrow \ell_o$.

Below, we state some properties of the reachability relation.

Lemma 7.2. (Properties of Reachability)

$$\textsc{ReachRefl}\quad \ell_n \rightsquigarrow \ell_n$$

$$\textsc{ReachTrans}\quad \frac{\ell_n \rightsquigarrow \ell_o \qquad \ell_o \rightsquigarrow \ell_m}{\ell_n \rightsquigarrow \ell_m}$$

$$\textsc{ReachPers}\quad \frac{\ell_n \rightsquigarrow \ell_m}{\square(\ell_n \rightsquigarrow \ell_m)}$$

$$\textsc{ReachNoCycles}\quad \frac{\ell_n \rightsquigarrow \ell_m \qquad \ell_n \rightsquigarrow \ell_m \qquad \ell_n \rightsquigarrow \ell_{\mathsf{dummy}} \qquad \ell_{\mathsf{dummy}} \mapsto_{1/2} \mathsf{inj}_1\ ()}{\ulcorner \ell_n = \ell_m \urcorner}$$

The lemma ReachNoCycles states that if two locations reach each other and reach the dummy location, then they must be the same location. In practice, this means that there are no cycles in the linked list. The assumption that they reach the dummy node is essential since it ensures that the linked list has an end-point.

Additionally, we define abstract reachability as in [VB21], using the resource algebra $Auth(\mathcal{P}(Loc))$ with disjoint union as the composition.

Definition 7.3. (Abstract Reachability and Tied-To)
First, we define *abstract reachability*, written $\ell_n \dashrightarrow \gamma_m$, for a location ℓ_n that can reach some ghost name γ_m. This proposition is persistent.

$$\ell_n \dashrightarrow \gamma_m \triangleq \boxed{\circ\{\ell_n\}}^{\gamma_m}.$$

The ghost name γ_m is *tied to* a physical location ℓ_m, written $\gamma_m \rightarrowtail \ell_m$. This assertion is *not* persistent.

$$\gamma_m \rightarrowtail \ell_m \triangleq \exists S,\ \boxed{\bullet S}^{\gamma_m} * \mathop{\Large *}_{\ell \in S} \ell_m \rightsquigarrow \ell.$$

The idea is that we can track reachability of locations, i.e. $\ell_n \rightsquigarrow \ell_m$, using abstract reachability, and vice versa. This is formalised in the following rules for abstract and physical reachability:

AbsReachAlloc
$$\Rightarrow\!\exists \gamma, \gamma \rightarrowtail l * l \dashrightarrow \gamma$$

AbsReachConcr
$$\frac{\ell_n \dashrightarrow \gamma_m \qquad \gamma_m \rightarrowtail \ell_m}{\ell_n \rightsquigarrow \ell_m * \gamma_m \rightarrowtail \ell_m}$$

ConcrReachAbs
$$\frac{\ell_n \rightsquigarrow \ell_m \qquad \gamma_m \rightarrowtail \ell_m}{\Rightarrow\!\ell_n \dashrightarrow \gamma_m * \gamma_m \rightarrowtail \ell_m}$$

AbsReachAdvance
$$\frac{\gamma_m \rightarrowtail \ell_m \qquad \ell_m \rightsquigarrow \ell_o}{\Rightarrow\!\gamma_m \rightarrowtail \ell_o * \ell_o \dashrightarrow \gamma_m}$$

For example, AbsReachConcr and ConcrReachAbs allow going from abstract to physical reachability and vice versa. The rule AbsReachAdvance lets us "advance" the abstract reachability by updating the physical location that the ghost name is tied to, if the new location is reachable from the previous one.

The main motivation for introducing abstract reachability is to track the global variables First, Last and the location pointing to the dummy node, and their position in the queue.
The abstract reachability relation and tied-to predicate should capture the following properties of the linked list representing the queue:

1. The dequeuer dequeues from the left of the queue. This means that the First pointer only moves to the right, towards the last node. Any node that is or has been the first node always reaches any current and future last node.
2. The enqueuer enqueues to the right, so the Last pointer also only moves to the right. Any node that is or has been the last node reaches any current or future the dummy node – possibly by reflexivity.
3. The de facto last location in the linked list, called ℓ_{dummy}, is the location that always points to the dummy node.

We encode the above properties using by tying the ghost names, γ_{First} and γ_{Last} to the locations pointed to by the global variables First and Last respectively, and additionally tying the name γ_{Dummy} to the location ℓ_{dummy}.
We define the absReach predicate used in the invariant to be exactly these properties:

$$\begin{aligned}\mathsf{absReach}(G, \ell_{\mathsf{First}}, \ell_{\mathsf{Last}}, \ell_{\mathsf{dummy}}) \triangleq\ & \gamma_{\mathsf{First}} \rightarrowtail \ell_{\mathsf{First}} * \ell_{\mathsf{First}} \dashrightarrow \gamma_{\mathsf{Last}} \\ & * \gamma_{\mathsf{Last}} \rightarrowtail \ell_{\mathsf{Last}} * \ell_{\mathsf{Last}} \dashrightarrow \gamma_{\mathsf{Dummy}} \\ & * \gamma_{\mathsf{Dummy}} \rightarrowtail \ell_{\mathsf{dummy}}.\end{aligned}$$

By keeping this in the queue invariant, we maintain sufficient information about the shape of the queue to update the First and Last pointers in the dequeue and enqueue operations.

We will go through the dequeue operation which necessitates the addition of abstract reachability in the invariant. In the proof of the dequeue specification, when we move the First pointer from the first location ℓ_{First} to the next location, ℓ_{next}, we prove that ℓ_{next} reaches the current last location, ℓ_{Last}.
This seems intuitively correct and easy to prove. However, without abstract reachability and tied-to assertions, we cannot prove it; then we could only make statements about the current first, last and dummy locations. The abstract reachability and tied-to assertions allow us to keep the information that e.g. any location that has *ever* been the last location can reach all future last and dummy locations.

7.5.1 Dequeue

We go through the implementation of the dequeue operation (5.3.3).
From the precondition, we get the assumption $\mathsf{isQueueConc}(q, G, \Phi)$ and the points-tos:

$$\mathsf{First} \mapsto_{1/2} \ell_{\mathsf{First}} * \mathsf{FreeLater} \mapsto \ell_{\mathsf{FL}} * \mathsf{isOptVal}(\mathsf{Help}, v^?_{\mathsf{Help}}) * \mathsf{isOptPts}(\ell_{\mathsf{FL}}, v^?_{\mathsf{FL}}).$$

The dequeuer owns (part of) the points-tos for $\mathsf{First}, \mathsf{FreeLater}, \mathsf{Help}$ and ℓ_{FL}, so by agreement of points-tos, we know that ℓ_{First}, ℓ_{FL}, $v^?_{\mathsf{Help}}$ and $v^?_{\mathsf{FL}}$ remain unchanged by the enqueuer throughout the program. Therefore, we do not need to take into account that they might change each time we open the invariant. The other existentially quantified variables on the other hand, e.g. ℓ_{Last}, xs and ℓ_{dummy}, might be updated by the enqueuer. Therefore, we write a superscript index on these variables for each new opening of the invariant to indicate that they might be different, i.e. $\ell^0_{\mathsf{Last}}, \ell^1_{\mathsf{Last}}, \ell^2_{\mathsf{Last}}$, etc.

We first load the current first location, ℓ_{First}.
We want to read the location stored in Last, so we open the invariant and get $\mathsf{Last} \mapsto_{1/2} \ell^0_{\mathsf{Last}}$ for some location ℓ^0_{Last}.
We keep (part of) the persistent proposition from $\mathsf{absReach}(G, \ell_{\mathsf{First}}, \ell^0_{\mathsf{Last}}, \ell^0_{\mathsf{dummy}})$ for later use, namely that the current last location reaches any current and future dummy and last location, and similarly for the current first location:

$$\square\,(\ell^0_{\mathsf{Last}} \dashrightarrow \gamma_{\mathsf{Dummy}}) * \square\,(\ell_{\mathsf{First}} \dashrightarrow \gamma_{\mathsf{Last}}).$$

Secondly, we prove $\square\,(\ell^0_{\mathsf{Last}} \dashrightarrow \gamma_{\mathsf{Last}})$. This follows from lemma ConcrReachAbs using $\gamma_{\mathsf{Last}} \rightarrowtail \ell^0_{\mathsf{Last}}$ and ReachRefl.
Finally, we prove that $\square\,(\ell_{\mathsf{First}} \rightsquigarrow \ell^0_{\mathsf{Last}})$ This follows immediately from the lemma AbsReachConcr, using $\ell_{\mathsf{First}} \dashrightarrow \gamma_{\mathsf{Last}}$ and $\gamma_{\mathsf{Last}} \rightarrowtail \ell^0_{\mathsf{Last}}$ from $\mathsf{absReach}(G, \ell_{\mathsf{First}}, \ell^0_{\mathsf{Last}}, \ell^0_{\mathsf{dummy}})$.
We close the invariant unchanged.

We now compare this location ℓ_{First} to the location stored in Last, ℓ^0_{Last}. There are two cases:

Either $\ell_{\mathsf{First}} = \ell^0_{\mathsf{Last}}$, which implies that the queue is empty. We return the unit value and give back the points-tos from the precondition to satisfy the postcondition.

Otherwise, $\ell_{\mathsf{First}} \neq \ell^0_{\mathsf{Last}}$. The next step is to load the value stored in ℓ_{First}.
Since the dequeuer does not own this points-to, we open the invariant. The current value of the queue elements is the list xs^1, the current last location is ℓ^1_{Last} and the current dummy location is ℓ^1_{dummy}.
We want to prove that the queue xs^1 is non-empty. If $xs^1 = \mathsf{nil}$ then by unfolding $\mathsf{isNodeList}(\ell_{\mathsf{First}}, \ell^1_{\mathsf{dummy}}, \mathsf{nil})$, we get $\ell_{\mathsf{First}} = \ell^1_{\mathsf{dummy}}$. In particular, from the assumption $\ell_{\mathsf{First}} \neq \ell^0_{\mathsf{Last}}$, we have $\ell^1_{\mathsf{dummy}} \neq \ell^0_{\mathsf{Last}}$. We want to reach a contradiction by proving that $\ell^1_{\mathsf{dummy}} = \ell^0_{\mathsf{Last}}$.

First, we prove that $\ell^0_{\mathsf{Last}} \rightsquigarrow \ell^1_{\mathsf{dummy}}$. We first conclude from $\mathsf{absReach}(G, \ell_{\mathsf{First}}, \ell^1_{\mathsf{Last}}, \ell^1_{\mathsf{dummy}})$ that the dummy ghost name is tied to ℓ^1_{dummy}, $\gamma_{\mathsf{Dummy}} \rightarrowtail \ell^1_{\mathsf{dummy}}$. Secondly, we own $\square\,(\ell^0_{\mathsf{Last}} \dashrightarrow \gamma_{\mathsf{Dummy}})$ from earlier. By applying lemma AbsReachConcr with these two facts, we conclude exactly $\ell^0_{\mathsf{Last}} \rightsquigarrow \ell^1_{\mathsf{dummy}}$.

Secondly, we prove that $\ell^1_{\mathsf{dummy}} \rightsquigarrow \ell^0_{\mathsf{Last}}$. This follows immediately from the earlier fact that $\ell_{\mathsf{First}} \rightsquigarrow \ell^0_{\mathsf{Last}}$ and the assumed equality $\ell_{\mathsf{First}} = \ell^1_{\mathsf{dummy}}$.

Finally, we know $\ell^1_{\mathsf{dummy}} \rightsquigarrow \ell^1_{\mathsf{dummy}}$ by ReachRefl, and $\ell^1_{\mathsf{dummy}} \mapsto_{1/2} \mathsf{inj}_1\ ()$ from the invariant.
By combining these four facts, we can apply lemma ReachNoCycles, which gives us the contradiction $\ell^1_{\mathsf{dummy}} = \ell^0_{\mathsf{Last}}$.

We conclude that list of queue elements is non-empty. Therefore, we know $xs^1 = (ts, tid, w) :: xs^1_{tail}$ for some ts, tid, w and xs^1_{tail}.
Furthermore, by unfolding $\mathsf{isNodeList}(\ell_{\mathsf{First}}, \ell^1_{\mathsf{dummy}}, (ts, tid, w) :: xs^1_{tail})$, we get two persistent a points-to assertions, $\ell_{\mathsf{First}} \mapsto_{\square} \mathsf{inj}_2\ ((ts, tid, w), \mathsf{next}) * \mathsf{next} \mapsto_{\square} \ell_{\mathsf{next}}$ for some locations next and ℓ_{next}.

We load the value $\mathsf{inj}_2\ ((ts, tid, w), \mathsf{next})$ and close the invariant unchanged.

Next, we want to store the value $\mathsf{inj}_2\ (ts, tid, w)$ in Help. We already own half of the points-to, $\mathsf{Help} \mapsto_{1/2} v^?_{\mathsf{Help}}$, so we open the invariant to get the other half. We store the value $\mathsf{inj}_2\ (ts, tid, w)$ in Help and close the invariant with the updated $v^?_{\mathsf{Help}}$.[2]
Now, we want to move the first pointer to the next node, i.e. store ℓ_{next} in First.

[2] It is in fact more involved to maintain the invariant around the help value, which we will explain in the following section 7.6. It is not our focus in this walk-through of the proof.

We open the invariant to get the other half of the points-to $\mathsf{First} \mapsto_{1/2} \ell_{\mathsf{First}}$. The current queue elements are xs^2, and the current last and dummy locations are ℓ^2_{Last} and ℓ^2_{dummy}.
We store ℓ_{next} in First, which gives us $\mathsf{First} \mapsto_{1/2} \ell_{\mathsf{next}}$ for the dequeuer and the other half for the invariant.

We now have to make sure that the new first location, ℓ_{next}, satisfies the properties described in the invariant:

First, we prove $\ell_{\mathsf{next}} \dashrightarrow \gamma_{\mathsf{Last}}$ using $\gamma_{\mathsf{First}} \rightarrowtail \ell_{\mathsf{First}}$ from $\mathsf{absReach}(G, \ell_{\mathsf{First}}, \ell^2_{\mathsf{Last}}, \ell^2_{\mathsf{dummy}})$. By ConcrReachAbs, it suffices to prove that $\ell_{\mathsf{next}} \rightsquigarrow \ell^2_{\mathsf{Last}}$.
To prove this, we first prove that $\ell^0_{\mathsf{Last}} \rightsquigarrow \ell^2_{\mathsf{Last}}$. We apply lemma AbsReachConcr with $\square\,(\ell^0_{\mathsf{Last}} \dashrightarrow \gamma_{\mathsf{Last}})$ from earlier and with $\gamma_{\mathsf{Last}} \rightarrowtail \ell^2_{\mathsf{Last}}$ from $\mathsf{absReach}(G, \ell_{\mathsf{First}}, \ell^2_{\mathsf{Last}}, \ell^2_{\mathsf{dummy}})$.
We use the persistent points-tos $\ell_{\mathsf{First}} \mapsto_\square \mathsf{inj}_2\,((ts, tid, w), \mathsf{next}) * \mathsf{next} \mapsto_\square \ell_{\mathsf{next}}$, the reachability of $\ell_{\mathsf{First}} \rightsquigarrow \ell^0_{\mathsf{Last}}$ and $\ell^0_{\mathsf{Last}} \rightsquigarrow \ell^2_{\mathsf{Last}}$ and the inequality $\ell_{\mathsf{First}} \neq \ell^0_{\mathsf{Last}}$ to prove $\ell_{\mathsf{next}} \rightsquigarrow \ell^2_{\mathsf{Last}}$:

NextReachesNewLast
$$\frac{\ell_{\mathsf{First}} \mapsto_\square \mathsf{inj}_2\,((ts, tid, w), \mathsf{next}) \qquad \mathsf{next} \mapsto_\square \ell_{\mathsf{next}} \qquad \ell_{\mathsf{First}} \rightsquigarrow \ell^0_{\mathsf{Last}} \qquad \ell^0_{\mathsf{Last}} \rightsquigarrow \ell^2_{\mathsf{Last}} \qquad \ell_{\mathsf{First}} \neq \ell^0_{\mathsf{Last}}}{\ell_{\mathsf{next}} \rightsquigarrow \ell^2_{\mathsf{Last}}}$$

The proof of lemma NextReachesNewLast follows from unfolding the reachability relations $\ell_{\mathsf{First}} \rightsquigarrow \ell^0_{\mathsf{Last}}$ and $\ell^0_{\mathsf{Last}} \rightsquigarrow \ell^2_{\mathsf{Last}}$.

Second, we update $\gamma_{\mathsf{First}} \rightarrowtail \ell_{\mathsf{First}}$ from $\mathsf{absReach}(G, \ell_{\mathsf{First}}, \ell^2_{\mathsf{Last}}, \ell^2_{\mathsf{dummy}})$ to $\gamma_{\mathsf{First}} \rightarrowtail \ell_{\mathsf{next}}$. By lemma AbsReachAdvance and using $\gamma_{\mathsf{First}} \rightarrowtail \ell_{\mathsf{First}}$, it suffices to prove that $\ell_{\mathsf{First}} \rightsquigarrow \ell_{\mathsf{next}}$ This immediate, since ℓ_{First} reaches ℓ_{next} in one step: $\ell_{\mathsf{First}} \mapsto_\square \mathsf{inj}_2\,((ts, tid, w), \mathsf{next}) * \mathsf{next} \mapsto_\square \ell_{\mathsf{next}}$.

Third, we prove that $xs = (ts, tid, w) :: xs^2_{tail}$ for some xs^2_{tail}, and $\mathsf{isNodeList}(\ell_{\mathsf{next}}, \ell^2_{\mathsf{dummy}}, xs^2_{tail})$, using the following lemma:

isNodeListNext
$$\frac{\mathsf{isNodeList}(\ell_{\mathsf{First}}, \ell^2_{\mathsf{dummy}}, xs^2) \qquad \ell_{\mathsf{First}} \mapsto_\square \mathsf{inj}_2\,((ts, tid, w), \mathsf{next}) \qquad \mathsf{next} \mapsto_\square \ell_{\mathsf{next}} \qquad \ell^2_{\mathsf{dummy}} \mapsto_{1/2} \mathsf{inj}_1\,()}{\exists xs^2_{tail},\ \ulcorner xs = (ts, tid, w) :: xs^2_{tail} \urcorner * \mathsf{isNodeList}(\ell_{\mathsf{next}}, \ell^2_{\mathsf{dummy}}, xs^2_{tail})}$$

We already own the propositions $\mathsf{isNodeList}(\ell_{\mathsf{First}}, \ell^2_{\mathsf{dummy}}, xs^2)$ and $\ell^2_{\mathsf{dummy}} \mapsto_{1/2} \mathsf{inj}_1\,()$ from the invariant, and we know $\ell_{\mathsf{First}} \mapsto_\square \mathsf{inj}_2\,((ts, tid, w), \mathsf{next}) * \mathsf{next} \mapsto_\square \ell_{\mathsf{next}}$ from earlier. We apply lemma isNodeListNext.

Thirdly, we split the predicate $\mathsf{forAll}(xs^2, \Phi)$ into a predicate for the head and the tail, $\Phi(w)$ and $\mathsf{forAll}(xs^2_{tail}, \Phi)$.

Finally, we can close the invariant by letting the new first location be to ℓ_{next} and the list of queue elements be xs^2_{tail}.

We only sketch the rest of the proof, as nothing interesting happens, since we do not free any nodes anything in this setting:

We open the invariant and read the announced location, $\ell^{?3}_{\mathsf{A}}$. We close the invariant unchanged.
We compare $\ell^{?3}_{\mathsf{A}}$ to $\mathsf{inj}_2\,(\ell_{\mathsf{First}})$.
Either, $\ell^{?3}_{\mathsf{A}} = \mathsf{inj}_2\,\ell_{\mathsf{First}}$. This means that the location ℓ_{First} is not necessarily safe to free. In the real algorithm, we would have to free the location stored in $\mathsf{FreeLater}$.
Instead, we simply store ℓ_{First} in $\mathsf{FreeLater}$ and return the value of the previous first node $\mathsf{inj}_2\,(ts, tid, w)$.
Since we own $\Phi(w)$ and the dequeuer's parts of the points-tos, we can satisfy the postcondition.

Otherwise, $\ell^{?3}_{\mathsf{A}} \neq \mathsf{inj}_2\,\ell_{\mathsf{First}}$. This means ℓ_{First} is safe to free. We skip freeing ℓ_{First} in this setting, and we return $\mathsf{inj}_2\,(ts, tid, w)$ and satisfy the postcondition as described above.

7.5.2 enqueue

In the proof of $\mathsf{enqueue}$, we must maintain the tied-to assertions when we update (1) the dummy location and (2) the last location.

1. In the intermediate step, where ℓ'_{dummy} replaces ℓ_{Last} as the dummy location, we know

$$\mathsf{absReach}(G, \ell^0_{\mathsf{First}}, \ell_{\mathsf{Last}}, \ell_{\mathsf{Last}})$$

for some ℓ^0_{First} from the invariant. We update $\gamma_{\mathsf{Dummy}} \rightarrowtail \ell_{\mathsf{Last}}$ to $\gamma_{\mathsf{Dummy}} \rightarrowtail \ell'_{\mathsf{dummy}} * \square(\ell'_{\mathsf{dummy}} \dashrightarrow \gamma_{\mathsf{Dummy}})$ using lemma AbsReachAdvance. It suffices to prove that $\ell_{\mathsf{Last}} \rightsquigarrow \ell'_{\mathsf{dummy}}$, which is immediate, since

$$\ell_{\mathsf{Last}} \mapsto_{\square} \mathsf{inj}_2\,((ts, tid, w), \mathsf{next}) * \mathsf{next} \mapsto_{\square} \ell'_{\mathsf{dummy}}.$$

This gives us $\mathsf{absReach}(G, \ell^0_{\mathsf{First}}, \ell_{\mathsf{Last}}, \ell'_{\mathsf{dummy}})$.

2. Similarly, when we update Last to point to the new dummy location ℓ'_{dummy} instead of ℓ_{Last}, we know

$$\mathsf{absReach}(G, \ell^1_{\mathsf{First}}, \ell_{\mathsf{Last}}, \ell'_{\mathsf{dummy}})$$

for some ℓ^1_{First} from the invariant. We update $\gamma_{\mathsf{Last}} \rightarrowtail \ell_{\mathsf{Last}}$ to $\gamma_{\mathsf{Last}} \rightarrowtail \ell'_{\mathsf{dummy}} * \square(\ell'_{\mathsf{dummy}} \dashrightarrow \gamma_{\mathsf{Last}})$ using lemma AbsReachAdvance. Again, it suffices to prove $\ell_{\mathsf{Last}} \rightsquigarrow \ell'_{\mathsf{dummy}}$, which is immediate. Lastly, we need to prove that the new last location, ℓ'_{dummy}, reaches (abstractly) the current dummy location, which is ℓ'_{dummy} itself. This follows from ConcrReachAbs and ReachRefl.

We conclude $\mathsf{absReach}(G, \ell^1_{\mathsf{First}}, \ell'_{\mathsf{dummy}}, \ell'_{\mathsf{dummy}})$.

7.6 One-shot RA for $v^?_{\mathsf{Help}}$

In the readFront operation for the enqueuer, readFrontEnq, the enqueuer reads the value stored in Help, $v^?_{\mathsf{Help}}$, if the location pointed to by First changes during the readFrontEnq operation. This is described in detail in the walk-through of the readFrontEnq program (section 2.1.3).
The value $v^?_{\mathsf{Help}}$ is an option, but $v^?_{\mathsf{Help}}$ can never be `None` if the First pointer has changed. To see why this is the case, we must consider the dequeue operation (section 2.1.2): The dequeuer dequeues the front node pointed to by ℓ_{First} by moving the First pointer to the location pointing to the next node, ℓ_{next}. However, before storing ℓ_{next} in First, the dequeuer stores the value of the element in the front node in Help. Therefore, to change location stored in First, the dequeuer must necessarily have stored a value in Help.

To capture this as an invariant of the queue, we realise that $v^?_{\mathsf{Help}} = \texttt{None}$ if and only if ℓ_{First} points to the *initial* dummy node from the initialisation of the queue (see fig. 6). We use the agreement resource algebra (defined in [JKJ+18] §4.3) over locations, $Ag(Loc)$, to remember the initial dummy location ℓ_{init}.

The agreement RA $Ag(Loc)$ satisfies the following rules:

$$\textsc{AgreeAlloc}\quad \Rightarrow\!\!\!\!\mid \exists \gamma_{\mathsf{init}}.\ \boxed{\mathsf{ag}(\ell)}^{\gamma_{\mathsf{init}}} \qquad \textsc{AgreeAgree}\quad \frac{\boxed{\mathsf{ag}(\ell)}^{\gamma_{\mathsf{init}}} \qquad \boxed{\mathsf{ag}(\ell')}^{\gamma_{\mathsf{init}}}}{\ulcorner \ell = \ell' \urcorner} \qquad \textsc{AgreePers}\quad \frac{\boxed{\mathsf{ag}(\ell)}^{\gamma_{\mathsf{init}}}}{\square \boxed{\mathsf{ag}(\ell)}^{\gamma_{\mathsf{init}}}}$$

Additionally, we use the one-shot RA (defined in [JKJ+18] §3.1), $\mathit{OneShot}(())$ to track the transition of $v^?_{\mathsf{Help}}$ from `None` to `Some`(v).

$$\textsc{PendingAlloc}\quad \Rightarrow\!\!\!\!\mid \exists \gamma_{\mathsf{help}}.\ \mathsf{pending}(\gamma_{\mathsf{help}}) \qquad \textsc{PendingUpdShot}\quad \frac{\mathsf{pending}(\gamma_{\mathsf{help}})}{\Rightarrow\!\!\!\!\mid \mathsf{shot}(\gamma_{\mathsf{help}})} \qquad \textsc{PendingShotFalse}\quad \frac{\mathsf{pending}(\gamma_{\mathsf{help}}) \qquad \mathsf{shot}(\gamma_{\mathsf{help}})}{\mathsf{False}} \qquad \textsc{ShotPers}\quad \frac{\mathsf{shot}(\gamma_{\mathsf{help}})}{\square\, \mathsf{shot}(\gamma_{\mathsf{help}})}$$

We keep the following resources in the invariant:

$$\mathsf{helpValOneShot}(\ell_{\mathsf{First}}, \ell_{\mathsf{init}}, v^?_{\mathsf{Help}}, \gamma_{\mathsf{init}}, \gamma_{\mathsf{help}}) \triangleq \boxed{\mathsf{ag}(\ell_{\mathsf{init}})}^{\gamma_{\mathsf{init}}} * \ell_{\mathsf{init}} \rightsquigarrow \ell_{\mathsf{First}} * \mathsf{helpDisj}(\ell_{\mathsf{First}}, \ell_{\mathsf{init}}, v^?_{\mathsf{Help}}, \gamma_{\mathsf{help}}).$$

We keep the initial dummy location ℓ_{init} and the fact that it always reaches the current first element of the queue (since the First pointer moves from left to right in the linked list). Additionally, we have two cases for the help value, $v^?_{\mathsf{Help}}$:

$$\begin{aligned}\mathsf{helpDisj}(\ell_{\mathsf{First}}, \ell_{\mathsf{init}}, v^?_{\mathsf{Help}}, \gamma_{\mathsf{help}}) \triangleq\ & \big(\mathsf{pending}(\gamma_{\mathsf{help}}) * \ulcorner v^?_{\mathsf{Help}} = \texttt{None} \urcorner * \ulcorner \ell_{\mathsf{First}} = \ell_{\mathsf{init}} \urcorner\big) \vee \\ & \big(\mathsf{shot}(\gamma_{\mathsf{help}}) * \exists ts, tid, w.\ \ulcorner v^?_{\mathsf{Help}} = \texttt{Some}(ts, tid, w) \urcorner\big).\end{aligned}$$

Either, the front node of the queue is still the initial dummy node ($\ulcorner \ell_{\mathsf{First}} = \ell_{\mathsf{init}} \urcorner$). In this case, we are still in the pending case, as the help value is `None`.
Otherwise, the help value has been updated to a value from the queue, and the dequeuer has performed the update from $\mathsf{pending}(\gamma_{\mathsf{help}})$ to $\mathsf{shot}(\gamma_{\mathsf{help}})$ (PendingUpdShot).

We will see how to use and maintain this in the invariant in the two affected operations:

7.6.1 readFrontEnq

We go through the readFrontEnq program (section 5.3.4).
We get $\mathsf{isQueueConc}(q, G, \Phi)$ as well as $\mathsf{Last} \mapsto_{1/2} \ell_{\mathsf{Last}} * \mathsf{Announce} \mapsto_{1/2} \ell^{?}_{\mathsf{A}} * \ell_{\mathsf{Last}} \mapsto_{1/2} \mathsf{inj}_1\ ()$ from the precondition.

We want to read the location stored in First, so we open the invariant and get $\mathsf{First} \mapsto_{1/2} \ell^0_{\mathsf{First}}$ for some location ℓ^0_{First}. We now compare this location ℓ^0_{First} to the location stored in Last, ℓ_{Last}.
There are two cases.

In the first case, $\ell^0_{\mathsf{First}} = \ell_{\mathsf{Last}}$. This implies that the queue is empty. We close the invariant unchanged, and return the unit value. We give back $\mathsf{Last} \mapsto_{1/2} \ell_{\mathsf{Last}} * \mathsf{Announce} \mapsto_{1/2} \ell^{?}_{\mathsf{A}} * \ell_{\mathsf{Last}} \mapsto_{1/2} \mathsf{inj}_1\ ()$ to satisfy the postcondition.

In the other case, $\ell^0_{\mathsf{First}} \neq \ell_{\mathsf{Last}}$, and we can conclude that the list of queue elements xs^0 is non-empty: If $xs^0 = \mathsf{nil}$ then by unfolding $\mathsf{isNodeList}(\ell^0_{\mathsf{First}}, \ell_{\mathsf{dummy}}, \mathsf{nil})$, we get $\ell^0_{\mathsf{First}} = \ell_{\mathsf{dummy}}$. Now, by the $\mathsf{lastDisj}(\ell_{\mathsf{Last}}, \ell_{\mathsf{dummy}})$, we have two cases. If we are in the LHS, then $\ell_{\mathsf{Last}} = \ell_{\mathsf{dummy}}$. This is a contradiction, since $\ell_{\mathsf{dummy}} = \ell^0_{\mathsf{First}} \neq \ell_{\mathsf{Last}}$. If we are in the RHS, then $\exists ts, tid, w, \mathsf{next}.\ \ell_{\mathsf{Last}} \mapsto_{\square} \mathsf{inj}_2\ ((ts, tid, w), \mathsf{next}) * \mathsf{next} \mapsto_{\square} \ell_{\mathsf{dummy}}$. This is a contradiction, since the enqueuer already owns $\ell_{\mathsf{Last}} \mapsto_{1/2} \mathsf{inj}_1\ ()$, and $\ell_{\mathsf{Last}} \mapsto_{\square} ((ts, tid, w), \mathsf{next}) * \ell_{\mathsf{Last}} \mapsto_{1/2} \mathsf{inj}_1\ () \vdash \mathsf{False}$.
Therefore, we know $xs^0 = (ts, tid, w) :: xs^0_{tail}$ for some ts, tid, w and xs'. Furthermore, by unfolding $\mathsf{isNodeList}(\ell^0_{\mathsf{First}}, \ell_{\mathsf{Last}}, (ts, tid, w) :: xs^0_{tail})$, we get a points-to assertion, $\ell^0_{\mathsf{First}} \mapsto_{\square} \mathsf{inj}_2\ ((ts, tid, w), \mathsf{next})$ for some location next.

We keep (part of) the persistent proposition from $\mathsf{helpValOneShot}(\ell^0_{\mathsf{First}}, \ell_{\mathsf{init}}, v^{?0}_{\mathsf{Help}}, \gamma_{\mathsf{init}}, \gamma_{\mathsf{help}})$ for later use, namely:

$$\square \left(\boxed{\mathsf{ag}(\ell_{\mathsf{init}})}^{\gamma_{\mathsf{init}}} * \left(\ulcorner \ell^0_{\mathsf{First}} = \ell_{\mathsf{init}} \urcorner \vee \mathsf{shot}(\gamma_{\mathsf{help}}) \right) \right).$$

We keep the ghost resource $\boxed{\mathsf{ag}(\ell_{\mathsf{init}})}^{\gamma_{\mathsf{init}}}$ to remember the value of ℓ_{init}. Additionally, we keep the disjunction stating that either ℓ^0_{First} is the initial dummy, or we are in the $\mathsf{shot}(\gamma_{\mathsf{help}})$ case, meaning that the help value cannot be `None`. We close the invariant unchanged.

We open the invariant and store $\mathsf{inj}_2\ (\ell^0_{\mathsf{First}})$ in Announce. We close the invariant again with the updated announced location.

We open the invariant again and load the (possibly changed) location ℓ^1_{First} stored in First. We have two cases:

In the first case, $\ell^1_{\mathsf{First}} = \ell^0_{\mathsf{First}}$. This means that the first node has not been dequeued since we loaded ℓ^0_{First}. We close the invariant unchanged.
Since $\ell^1_{\mathsf{First}} = \ell^0_{\mathsf{First}}$, we have to load the value stored in the node that ℓ^0_{First} points to.
We own the persistent points-to $\ell^0_{\mathsf{First}} \mapsto_{\square} \mathsf{inj}_2\ ((ts, tid, w), \mathsf{next})$, so we load and return the value (ts, tid, w).
We give back the points-tos $\mathsf{Last} \mapsto_{1/2} \ell_{\mathsf{Last}} * \mathsf{Announce} \mapsto_{1/2} \mathsf{inj}_2\ \ell_{\mathsf{First}} * \ell_{\mathsf{Last}} \mapsto_{1/2} \mathsf{inj}_1\ ()$ in the precondition.

In the other case, $\ell^1_{\mathsf{First}} \neq \ell^0_{\mathsf{First}}$. This means that the first node has been or is in the process of being dequeued since we loaded ℓ^0_{First}. Therefore, and ℓ^0_{First} might not be safe to read.
Before we close the invariant, we need two persistent facts:
First, keep the persistent reachability proposition from $\mathsf{helpValOneShot}(\ell^1_{\mathsf{First}}, \ell_{\mathsf{init}}, v^{?\prime}_{\mathsf{Help}}, \gamma_{\mathsf{init}}, \gamma_{\mathsf{help}})$ for later use: $\square(\ell_{\mathsf{init}} \rightsquigarrow \ell^1_{\mathsf{First}})$. Note that ℓ_{init} has not changed from the last opening of the invariant because of AgreeAgree.

Secondly, we prove the persistent proposition $\Box\,(\ell^1_{\mathsf{First}} \dashrightarrow \gamma_{\mathsf{First}})$, using $\mathsf{absReach}(G, \ell^1_{\mathsf{First}}, \ell_{\mathsf{Last}}, \ell_{\mathsf{dummy}})$ and lemma ConcrReachAbs and ReachRefl. This means that location ℓ^1_{First} can reach any future first location.
We close the invariant unchanged.

Next, we must read the value stored in Help. We open the invariant and we get $\mathsf{isOptVal}(\mathsf{Help}, v^{?2}_{\mathsf{Help}})$ for some option value $v^{?2}_{\mathsf{Help}}$. Additionally, we get the $\mathsf{helpDisj}(\ell^2_{\mathsf{First}}, \ell_{\mathsf{init}}, v^{?2}_{\mathsf{Help}}, \gamma_{\mathsf{help}})$ and $\ell_{\mathsf{init}} \rightsquigarrow \ell^2_{\mathsf{First}}$ for the current first location ℓ^2_{First}.

We now have two cases, depending on the value of $v^{?2}_{\mathsf{Help}}$:
In the first case, $v^{?2}_{\mathsf{Help}} = \texttt{Some}(ts, tid, w)$. In this case, we load (ts, tid, w), close the invariant unchanged, and return the value $\mathsf{inj}_2\ (ts, tid, w)$. We give back the points-tos $\mathsf{Last} \mapsto_{1/2} \ell_{\mathsf{Last}} * \mathsf{Announce} \mapsto_{1/2} \mathsf{inj}_2\ \ell^0_{\mathsf{First}} * \ell_{\mathsf{Last}} \mapsto_{1/2} \mathsf{inj}_1\ ()$ in the precondition.

In the other case, $v^{?2}_{\mathsf{Help}} = \texttt{None}$, we want to reach a contradiction, since – as argued – the help value can never be None if the first location has changed. We consider the two cases of $\mathsf{helpDisj}(\ell^2_{\mathsf{First}}, \ell_{\mathsf{init}}, v^{?2}_{\mathsf{Help}}, \gamma_{\mathsf{help}})$ from the invariant, and aim to reach a contradiction in both cases.

* $\mathsf{pending}(\gamma_{\mathsf{help}}) * \ulcorner v^{?2}_{\mathsf{Help}} = \texttt{None} \urcorner * \ulcorner \ell^2_{\mathsf{First}} = \ell_{\mathsf{init}} \urcorner$:

  We consider the two cases of $\Box\ \left(\ulcorner \ell^0_{\mathsf{First}} = \ell_{\mathsf{init}} \urcorner \vee \mathsf{shot}(\gamma_{\mathsf{help}})\right)$ from earlier.

  If we are in the RHS, then we reach a contradiction by lemma PendingShotFalse, since we own both $\mathsf{shot}(\gamma_{\mathsf{help}})$ and $\mathsf{pending}(\gamma_{\mathsf{help}})$.

  Therefore, we must be in the case that $\ell^2_{\mathsf{First}} = \ell^0_{\mathsf{First}} = \ell_{\mathsf{init}}$. We note that we have assumed $\ell^0_{\mathsf{First}} \neq \ell^1_{\mathsf{First}}$. In particular, $\ell_{\mathsf{init}} \neq \ell^1_{\mathsf{First}}$.

  We want to reach a contradiction by showing that ℓ^1_{First} has to be equal to the initial dummy location ℓ_{init}, since we have assumed that both the earlier first location ℓ^0_{First} and the current first location ℓ^2_{First} are equal to ℓ_{init}. Intuitively, is not possible for the first location to point to the initial dummy node after it has moved to point to the next node.

  First, we prove that $\ell^1_{\mathsf{First}} \rightsquigarrow \ell_{\mathsf{init}}$.

  We first conclude from $\mathsf{absReach}(G, \ell^2_{\mathsf{First}}, \ell_{\mathsf{Last}}, \ell_{\mathsf{dummy}})$ that the first ghost name is tied to $\ell^2_{\mathsf{First}} = \ell_{\mathsf{init}}$, i.e. $\gamma_{\mathsf{First}} \rightarrowtail \ell^2_{\mathsf{First}}$, . Secondly, we own $\Box(\ell_{\mathsf{init}} \rightsquigarrow \ell^1_{\mathsf{First}})$ from earlier. By applying lemma AbsReachConcr with these two facts, we conclude exactly $\ell^1_{\mathsf{First}} \rightsquigarrow \ell^2_{\mathsf{First}}$, i.e. $\ell^1_{\mathsf{First}} \rightsquigarrow \ell_{\mathsf{init}}$.

  Secondly, we prove that $\ell_{\mathsf{init}} \rightsquigarrow \ell_{\mathsf{dummy}}$. This follows from $\mathsf{absReach}(G, \ell^2_{\mathsf{First}}, \ell_{\mathsf{Last}}, \ell_{\mathsf{dummy}})$ and applying lemma AbsReachConcr twice to first conclude $\ell_{\mathsf{init}} = \ell^2_{\mathsf{First}} \rightsquigarrow \ell_{\mathsf{Last}}$ and then $\ell_{\mathsf{Last}} \rightsquigarrow \ell_{\mathsf{dummy}}$. We conclude by ReachTrans.

  Finally, we know $\Box(\ell_{\mathsf{init}} \rightsquigarrow \ell^1_{\mathsf{First}})$ from earlier.

  These three facts, $\ell^1_{\mathsf{First}} \rightsquigarrow \ell_{\mathsf{init}}$, $\ell_{\mathsf{init}} \rightsquigarrow \ell_{\mathsf{dummy}}$ and $\ell_{\mathsf{init}} \rightsquigarrow \ell^1_{\mathsf{First}}$, together with the points-to $\ell_{\mathsf{dummy}} \mapsto_{1/2} \mathsf{inj}_1\ ()$, give us the contradiction $\ell_{\mathsf{init}} = \ell^1_{\mathsf{First}}$. This follows immediately from ReachNoCycles.

* $\mathsf{shot}(\gamma_{\mathsf{help}}) * \exists ts, tid, w. \ulcorner v^{?2}_{\mathsf{Help}} = \texttt{Some}(ts, tid, w) \urcorner$: This is a contradiction, since $v^{?2}_{\mathsf{Help}} = \texttt{None} \neq \texttt{Some}(ts, tid, w)$.

7.6.2 dequeue

We focus on the interaction with the $v^{?}_{\mathsf{Help}}$ in the proof of the specification for dequeue:

In the dequeue program (section 5.3.4), the dequeuer stores the value of the first node that is currently being dequeued, $\ell_{\mathsf{First}} \mapsto_{\Box} \mathsf{inj}_2\ ((ts, tid, w), \mathsf{next})$, in Help.

The dequeuer opens the invariant and combines its own part of the points-to $\mathsf{Help} \mapsto_{1/2} v^{?}_{\mathsf{Help}}$ with the other half in the invariant.
The dequeuer stores the value $\mathsf{inj}_2\,(ts, tid, w)$ in Help, and ensures that the one-shot resource has been updated to $\square\,\mathsf{shot}(\gamma_{\mathsf{help}})$.
We consider the two cases of the $\mathsf{helpDisj}$. Either, the help value was already $\mathtt{Some}(v)$ and $\mathsf{pending}(\gamma_{\mathsf{help}})$ has already been update to $\mathsf{shot}(\gamma_{\mathsf{help}})$, in which case we do nothing.
Otherwise, $v^{?}_{\mathsf{Help}} = \mathtt{None}$ and we own $\mathsf{pending}(\gamma_{\mathsf{help}})$, in which case we update $\mathsf{pending}(\gamma_{\mathsf{help}})$ to $\mathsf{shot}(\gamma_{\mathsf{help}})$ using PendingUpdShot.
We close the invariant by satisfying the RHS of the $\mathsf{helpDisj}$ with the new help value $\mathsf{inj}_2\,(ts, tid, w)$ and with $\mathsf{shot}(\gamma_{\mathsf{help}})$.

In the next step of the program, we store ℓ_{next} in First (where $\ell_{\mathsf{First}} \mapsto_{\square} \mathsf{inj}_2\,((ts, tid, w), \mathsf{next}) * \mathsf{next} \mapsto_{\square} \ell_{\mathsf{next}}$), and we close the invariant with ℓ_{next} as the new first location.
Among other things, we have to prove the $\mathsf{helpDisj}$ for ℓ_{next}, i.e. $\mathsf{helpDisj}(\ell_{\mathsf{next}}, \ell_{\mathsf{init}}, v^{?1}_{\mathsf{Help}}, \gamma_{\mathsf{help}})$, for some $v^{?1}_{\mathsf{Help}}$. We own $\mathsf{shot}(\gamma_{\mathsf{help}})$ from earlier, and since the dequeuer owns half of the points-to for Help, we know that $v^{?1}_{\mathsf{Help}} = \mathtt{Some}(ts, tid, w)$. With this, we satisfy the RHS of the disjunction:

$$\mathsf{shot}(\gamma_{\mathsf{help}}) * \exists ts, tid, w.\, \ulcorner v^{?1}_{\mathsf{Help}} = \mathtt{Some}(ts, tid, w) \urcorner.$$

8 Concurrent Specifications & Proofs (With Freeing)

In this section, we prove a concurrent specification for the full SESD queue algorithm, in which the dequeuer frees memory during the dequeue operation. This mainly affects the dequeue operation, in which the dequeuer frees memory, and the readFrontEnq operation, in which the enqueuer announces the location it wants to read, so that the dequeuer does not free it.
Freeing locations adds a lot more complexity to the concurrent reasoning about the queue in terms of communcation between the enqueuer and dequeuer, compared to the concurrent setting without freeing (section 7).

The specification that we prove in this setting is exactly the same as in the previous setting (section 7.3). The only difference is that the isEnqueuer and isDequeuer predicates are changed because of the additions to the queue predicate, isQueueConcFreeing.

In the previous section, we built the linked list that represents the queue using persistent points-to assertions. This allowed both the enqueuer and dequeuer to read the queue elements concurrently, but it also meant that nodes could never be freed. In this setting, we have to come up with a different representation of the linked list that does *not* use persistent points-tos.
Therefore, we introduce logical points-tos: A logical points-to represents a node in the queue, and it can be exchanged for a normal points-to, *as long as the points-to has not been freed*. Therefore, the main challenge is to capture exactly when a points-to has or has not been freed.

The dequeuer's task is mainly to ensure that it only frees a points-to when it is safe to do so, i.e. when the enqueuer will not try to read it. The enqueuer should, however, always be able to conclude that the location it wants to read in the readFrontEnq program has not been freed by the dequeuer. Otherwise, the program is not safe. Consequently, the main challenge in this setting is to characterise which locations in the linked list are safe to read and which are safe to free. The communication between the enqueuer and dequeuer happens via an updated queue invariant that includes additional ghost resources.

We introduce the following differences compared to the concurrent setting *without* freeing (section 7):

1. We store the logical points-tos in a points-to map, M, in section 8.2 and define the set of locations in the queue which have not yet been freed. The map M contains all the points-tos that make up the linked list representing the queue, including the freed ones. To make the set of not-freed locations precise, we introduce a location to-be-freed, tbf, that is used by the dequeuer to track which location is a candidate to be freed or is currently in the process of being freed. The map of points-tos is our replacement of the persistent points-tos we used earlier.

2. In section 8.2.1, we update the reachability relation from section 7.5 to only include locations in the queue, i.e. the locations with points-tos in M. Furthermoer, we define a *pure* reachability relation for locations in the map M, including the intermediate locations stored in the nodes.

   The motivation for adding these new reachability relations is to define the set of not-freed locations. In particular, any location that the current first node can reach has not been freed.

3. We include extra ghost resources to the invariant for communication about the announced location between the enqueuer and the dequeuer (section 8.4). The purpose of the announcement is to guarantee that the dequeuer never frees a node that the enqueuer might read.

In the following sections, we explain these new concepts detail and show how they are used to ensure safety of the algorithm.

8.1 Invariant

The new queue invariant is an extension of the queue invariant described in section 7.1.1 for the version of the algorithm that did not include freeing memory.

We first define the ghost names used in the invariant:

$$G \triangleq (\ \gamma_{\mathsf{First}}, \gamma_{\mathsf{Last}}, \gamma_{\mathsf{Dummy}}, \quad \text{(Abstract Reachability)}$$
$$\gamma_{\mathsf{help}}, \gamma_{\mathsf{init}} \quad \text{(One-Shot RA for Help)}$$
$$\gamma_{\mathsf{dom}}, \gamma_{\mathsf{pt}}, \gamma_{\mathsf{tbf}} \quad \text{(Points-To Map)}$$
$$\gamma_{\mathsf{FstAtAnno}}, \gamma_{\mathsf{pastAnnos}}, \gamma_{\mathsf{X}}, \gamma_{\mathsf{not\text{-}fstAtAnno}}\ ). \quad \text{(Announce)}$$

$$\mathsf{queueInv}(\mathsf{First}, \mathsf{Last}, \mathsf{FreeLater}, \mathsf{Announce}, \mathsf{Help}, G, \Phi) \triangleq$$
$$\exists \ell_{\mathsf{First}}, \ell_{\mathsf{Last}}, \ell_{\mathsf{A}}^?, \ell_{\mathsf{FL}}, v_{\mathsf{Help}}^?, \ell_{\mathsf{dummy}}, \ell_{\mathsf{init}}, xs, \mathsf{tbf}, \ell_{\mathsf{fstAtAnno}}^?, \ell_{\mathsf{X\text{-}A}}^?, \ell_{\mathsf{X\text{-}fstAtAnno}}^?, \ell_{\mathsf{not\text{-}fstAtAnno}}, \mathsf{S}_{pastAnnos}.$$
$$\mathsf{First} \mapsto_{1/2} \ell_{\mathsf{First}} * \mathsf{Last} \mapsto_{1/2} \ell_{\mathsf{Last}} *$$
$$\mathsf{isNodeList}(\ell_{\mathsf{First}}, \ell_{\mathsf{dummy}}, xs, \gamma_{\mathsf{pt}}) * \mathsf{forAll}(xs, \Phi) *$$
$$\ell_{\mathsf{dummy}} \mapsto_{1/2} \mathsf{inj}_1\ () * \boxed{\circ\{\ell_{\mathsf{dummy}}\}}^{\gamma_{\mathsf{dom}}} * \mathsf{lastDisj}(\ell_{\mathsf{Last}}, \ell_{\mathsf{dummy}}, \gamma_{\mathsf{pt}}) *$$
$$\mathsf{Announce} \mapsto_{1/2} \ell_{\mathsf{A}}^? * \mathsf{Help} \mapsto_{1/2} v_{\mathsf{Help}}^? * \mathsf{FreeLater} \mapsto_{1/2} \ell_{\mathsf{FL}} *$$
$$\mathsf{absReach}(\gamma_{\mathsf{First}}, \gamma_{\mathsf{Last}}, \gamma_{\mathsf{Dummy}}, \ell_{\mathsf{First}}, \ell_{\mathsf{Last}}, \ell_{\mathsf{dummy}}) *$$
$$\mathsf{helpValOneShot}(\gamma_{\mathsf{help}}, \gamma_{\mathsf{init}}, \ell_{\mathsf{First}}, \ell_{\mathsf{init}}, v_{\mathsf{Help}}^?, \gamma_{\mathsf{dom}}, \gamma_{\mathsf{pt}}) *$$
$$\mathsf{logicalPts}(\gamma_{\mathsf{pt}}, \gamma_{\mathsf{dom}}, \ell_{\mathsf{FL}}, \ell_{\mathsf{First}}, \mathsf{tbf}) *$$
$$\mathsf{ptNoneOrPtPersReach}(\ell_{\mathsf{FL}}, \ell_{\mathsf{First}}, \gamma_{\mathsf{dom}}, \gamma_{\mathsf{pt}}) *$$
$$\boxed{\tfrac{1}{2}(\mathsf{tbf})}^{\mathsf{freed}} * \mathsf{tbfDisj}(\mathsf{tbf}, \ell_{\mathsf{First}}, \ell_{\mathsf{FL}}, \gamma_{\mathsf{dom}}, \gamma_{\mathsf{pt}}) *$$
$$\boxed{\tfrac{1}{2}(\ell_{\mathsf{X\text{-}A}}^?, \ell_{\mathsf{X\text{-}fstAtAnno}}^?)}^{\gamma_{\mathsf{X}}} * \boxed{\tfrac{1}{2}(\ell_{\mathsf{fstAtAnno}}^?)}^{\gamma_{\mathsf{FstAtAnno}}} *$$
$$\boxed{\tfrac{1}{2}(\ell_{\mathsf{not\text{-}fstAtAnno}})}^{\gamma_{\mathsf{not\text{-}fstAtAnno}}} * \ulcorner \ell_{\mathsf{First}} \neq \ell_{\mathsf{not\text{-}fstAtAnno}} \urcorner *$$
$$\mathsf{fstAtAnnoDisj}(\ell_{\mathsf{fstAtAnno}}^?, \ell_{\mathsf{A}}^?, \ell_{\mathsf{X\text{-}A}}^?, \ell_{\mathsf{X\text{-}fstAtAnno}}^?, \ell_{\mathsf{First}}, \ell_{\mathsf{not\text{-}fstAtAnno}}) *$$
$$\mathsf{fstAtAnnoAnnoImpl}(\ell_{\mathsf{fstAtAnno}}^?, \ell_{\mathsf{A}}^?, \ell_{\mathsf{First}}, \mathsf{tbf}, \ell_{\mathsf{FL}}, \gamma_{\mathsf{dom}}, \gamma_{\mathsf{pt}}) *$$
$$\boxed{\bullet \mathsf{S}_{pastAnnos}}^{\gamma_{\mathsf{pastAnnos}}} * \mathsf{pastAnnosDisj}(\mathsf{S}_{pastAnnos}, \ell_{\mathsf{A}}^?, \ell_{\mathsf{First}}, \gamma_{\mathsf{pastAnnos}}, \gamma_{\mathsf{dom}}, \gamma_{\mathsf{pt}}).$$

8.2 A Map of Logical Points-tos

We introduce a map of logical points-to assertions. The idea is to mimic the points-tos we used in the non-freeing version, i.e. persistent poinsts-tos for non-empty nodes and intermediate locations, and half of a points-to for the location pointing to the dummy node. Unlike the normal persistent points-tos, we want to be able to free these logical points-tos. To achieve this, we introduce a points-to map M, mapping locations to either $\mathsf{inj}_1\ ()$ (not persistent) or to a value (persistent); in practice, the value is either a pair $\mathsf{inj}_2\ (v, \ell)$ or just a location $\mathsf{inj}_2\ \ell$). These are the three types of points-tos that make up the linked list representing the queue (see fig. 5.2). We define the map using the resource algebra $Auth(gmap(Loc, Ag(Val) +_i Ex()))$, and we keep the authoritative part of it, $\boxed{\bullet \mathsf{M}}^{\gamma_{\mathsf{pt}}}$, in the invariant.

These logical points-tos can be exchanged for real points-tos by providing proof that the location has not been freed. That is, we store the real points-tos of the logical points in M in the invariant, and we define a set $\mathsf{S}_{\mathsf{notFreed}}$ that captures exactly the set of locations in the domain of M whose real points-tos have not (yet) been freed.

We define the two different logical points-to predicates, called ptNone and ptPers:

$$\ell \xmapsto{\gamma_{\mathsf{pt}}}_{1/2} \mathsf{inj}_1\ () \triangleq \boxed{\circ\{\ell \mapsto \mathsf{inr}(\mathsf{ex}())\}}^{\gamma_{\mathsf{pt}}}, \quad \text{(ptNone)}$$
$$\ell \xmapsto{\gamma_{\mathsf{pt}}}_{\square} \mathsf{inj}_2\ (v) \triangleq \boxed{\circ\{\ell \mapsto \mathsf{inl}(\mathsf{ag}(v))\}}^{\gamma_{\mathsf{pt}}}. \quad \text{(ptPers)}$$

These points-to behave similarly to normal points-tos (Lemma 3.1):

Lemma 8.1. (Properties of logical points-to assertions)

$$\textsc{PtPersPers}\quad \frac{\ell \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{\square} \mathsf{inj}_2\,(v)}{\square(\ell \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{\square} \mathsf{inj}_2\,(v))} \qquad \textsc{PtPersAgree}\quad \frac{\ell \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{\square} \mathsf{inj}_2\,(v) \qquad \ell \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{\square} \mathsf{inj}_2\,(v')}{\ulcorner v = v' \urcorner} \qquad \textsc{PtNoneExcl}\quad \frac{\ell \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{1/2} \mathsf{inj}_1\,() \qquad \ell \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{1/2} \mathsf{inj}_1\,()}{\mathsf{False}}$$

$$\textsc{PtPersPtNoneExcl}\quad \frac{\ell \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{\square} \mathsf{inj}_2\,(v) \qquad \ell \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{1/2} \mathsf{inj}_1\,()}{\mathsf{False}}$$

The ptPers is persistent (PtPersPers), and ptNone is exclusive (PtNoneExcl), even though it represents one half of a normal points-to. The reason is that we will split a normal points-to $\ell \mapsto \mathsf{inj}_1\,()$ into $\ell \mapsto_{1/2} \mathsf{inj}_1\,() * \ell \mapsto_{1/2} \mathsf{inj}_1\,()$ and give one half to the logicalPts in exchange for $\ell \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{1/2} \mathsf{inj}_1\,()$. Therefore, there can only exists one ptNone per location.

The map of points-tos M is a map from locations to option values, $\mathit{gmap}(\mathit{Loc}, \mathit{option}(\mathit{Val}))$. We represent ptNone, $\ell \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{\square} \mathsf{inj}_2\,(v)$, as $[\ell \mapsto \mathtt{Some}(\mathsf{inj}_2\,(v))]$, and we represent ptPers, $\ell \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{1/2} \mathsf{inj}_1\,()$, as $[\ell \mapsto \mathtt{None}]$ in the map.[3]

We either store a full points-to (ptPers) or half of a points-to (ptNone) in the map, unless the location has been freed, in which case we store $\mathsf{freed}(\ell)$:

Definition 8.2. (ptMapPts)

$$\begin{aligned}&\mathsf{ptMapPts}(\mathsf{S}_{\mathsf{notFreed}}, \ell, v^?) \triangleq \\ &\Big(\exists v,\ \ulcorner v^? = \mathtt{Some}(v) \urcorner * \big((\ulcorner \ell \in \mathsf{S}_{\mathsf{notFreed}} \urcorner * \ell \mapsto v) \lor (\ulcorner \ell \notin \mathsf{S}_{\mathsf{notFreed}} \urcorner * \mathsf{freed}(\ell))\big)\Big) \\ &\lor \\ &\Big(\ulcorner v^? = \mathtt{None} \urcorner * \big((\ulcorner \ell \in \mathsf{S}_{\mathsf{notFreed}} \urcorner * \ell \mapsto_{1/2} \mathsf{inj}_1\,()) \lor (\ulcorner \ell \notin \mathsf{S}_{\mathsf{notFreed}} \urcorner * \mathsf{freed}(\ell))\big)\Big)\end{aligned}$$

For example, by owning e.g. $\ell \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{\square} \mathsf{inj}_2\,(v)$ we can access the real points-to $\ell \mapsto \mathsf{inj}_2\,(v)$, if we can prove that $\ell \in \mathsf{S}_{\mathsf{notFreed}}$.

In the next sections (8.2.1 and 8.2.2) we will define the set $\mathsf{S}_{\mathsf{notFreed}}$ such that it includes exactly the locations in the linked list representing the queue that have not been freed.

8.2.1 (Logical) M-Reachability

One of the ways to ensure that a location ℓ has not yet been freed is if the current first location ℓ_{First} can reach it.
The reason is the following: in dequeue, the dequeuer first updates the first location from ℓ_{First} to ℓ_{next}, and afterwards frees either ℓ_{First} or the location ℓ_{FL} stored in FreeLater. Neither of these locations — ℓ_{First} and ℓ_{FL} — are reachable from the updated first location ℓ_{next}. Intuitively, all locations reachable from the current first location are still part of the queue, and therefore cannot yet have be freed.

Since we are interested in reachability between nodes in the queue, we keep track of the locations whose logical points-tos are in the points-to map M by storing the domain of M in the invariant. For this, we use the RA $\mathit{Auth}(\mathit{gset}(\mathit{Loc}))$, and we keep the authoritative and fragmental views of the domain in the invariant: $\boxed{\bullet(\mathsf{dom}(\mathsf{M}))}^{\gamma_{\mathsf{dom}}} * \boxed{\circ(\mathsf{dom}(\mathsf{M}))}^{\gamma_{\mathsf{dom}}}$.
Using this RA, we can prove that a location ℓ is in the map M by owning $\boxed{\circ\{\ell\}}^{\gamma_{\mathsf{dom}}}$:

$$\boxed{\circ\{\ell\}}^{\gamma_{\mathsf{dom}}} \vdash \ulcorner \ell \in \mathsf{dom}(\mathsf{M}) \urcorner.$$

Keeping the fragmental view of the domain allows us to conclude $\boxed{\circ\{\ell\}}^{\gamma_{\mathsf{dom}}}$ from either $\ell \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{\square} \mathsf{inj}_2\,(v)$ or $\ell \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{1/2} \mathsf{inj}_1\,()$ *without* an update modality. This will be important in the proofs, although we will not go

[3] In practice, we use a function to transform this map into the RA map, mapping $\mathtt{Some}(v)$ to $\mathsf{inl}(\mathsf{ag}(v))$ and None to $\mathsf{inr}(\mathsf{ex}())$.

into detail in this report.

We first update the old definition of reachability (definition 7.1 in section 7.5). We replace the normal persistent points-tos with the new logical points-tos, and we additionally ensure that all locations are in $\mathsf{dom}(\mathsf{M})$. We call this updated reachability relation *logical M-reachability*:

Definition 8.3. (Logical M-Reachability)

$$\ell_n \overset{\gamma_{\mathsf{dom}},\gamma_{\mathsf{pt}}}{\rightsquigarrow} \ell_m \triangleq \left(\ulcorner \ell_n = \ell_m \urcorner * \boxed{\circ\{\ell_n\}}^{\gamma_{\mathsf{dom}}}\right) \vee \left(\exists ts, tid, w, \mathsf{next}, \ell_o.\ \ell_n \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{\square} \mathtt{Some}((ts, tid, w), \mathsf{next}) * \mathsf{next} \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{\square} \ell_o * \ell_o \overset{\gamma_{\mathsf{dom}},\gamma_{\mathsf{pt}}}{\rightsquigarrow} \ell_m\right).$$

This definition satisfies properties similar to the reachability relation from the non-freeing version, except for the highlighted changes:

Lemma 8.4. (Properties of Logical M-Reachability)

LogMReachRefl
$$\frac{\ulcorner \ell_n \in \mathsf{dom}(\mathsf{M}) \urcorner}{\ell_n \overset{\gamma_{\mathsf{dom}},\gamma_{\mathsf{pt}}}{\rightsquigarrow} \ell_n}$$

LogMReachTrans
$$\frac{\ell_n \overset{\gamma_{\mathsf{dom}},\gamma_{\mathsf{pt}}}{\rightsquigarrow} \ell_o \qquad \ell_o \overset{\gamma_{\mathsf{dom}},\gamma_{\mathsf{pt}}}{\rightsquigarrow} \ell_m}{\ell_n \overset{\gamma_{\mathsf{dom}},\gamma_{\mathsf{pt}}}{\rightsquigarrow} \ell_m}$$

LogMReachPers
$$\frac{\ell_n \overset{\gamma_{\mathsf{dom}},\gamma_{\mathsf{pt}}}{\rightsquigarrow} \ell_m}{\square(\ell_n \overset{\gamma_{\mathsf{dom}},\gamma_{\mathsf{pt}}}{\rightsquigarrow} \ell_m)}$$

LogMReachNoCycles
$$\frac{\ell_n \overset{\gamma_{\mathsf{dom}},\gamma_{\mathsf{pt}}}{\rightsquigarrow} \ell_m \qquad \ell_n \overset{\gamma_{\mathsf{dom}},\gamma_{\mathsf{pt}}}{\rightsquigarrow} \ell_m \qquad \ell_n \overset{\gamma_{\mathsf{dom}},\gamma_{\mathsf{pt}}}{\rightsquigarrow} \ell_{\mathsf{dummy}} \qquad \ell_{\mathsf{dummy}} \mapsto_{1/2} \mathtt{inj}_1\ () \qquad \boxed{\circ\{\ell_{\mathsf{dummy}}\}}^{\gamma_{\mathsf{dom}}}}{\ulcorner \ell_n = \ell_m \urcorner}$$

LogMReachNoCyclesPtNone
$$\frac{\ell_n \overset{\gamma_{\mathsf{dom}},\gamma_{\mathsf{pt}}}{\rightsquigarrow} \ell_m \qquad \ell_n \overset{\gamma_{\mathsf{dom}},\gamma_{\mathsf{pt}}}{\rightsquigarrow} \ell_m \qquad \ell_n \overset{\gamma_{\mathsf{dom}},\gamma_{\mathsf{pt}}}{\rightsquigarrow} \ell_{\mathsf{dummy}} \qquad \ell_{\mathsf{dummy}} \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{1/2} \mathtt{inj}_1\ ()}{\ulcorner \ell_n = \ell_m \urcorner}$$

The two logical-M-versions of the old lemma ReachNoCycles, LogMReachNoCycles and LogMReachNoCyclesPtNone, both additionally require proof that the dummy location is in the logical points-to map ($\ell \in \mathsf{M}$), either by providing $\boxed{\circ\{\ell_{\mathsf{dummy}}\}}^{\gamma_{\mathsf{dom}}}$ or by providing the logical points-to, ptNone, instead of the normal points-to.

We define one last reachability relation, called M-reachability. Unlike logical reachability, this relation is pure, i.e. defined in the meta-logic, not as an Iris proposition. The motivation for defining M-reachability is that we use it to characterise not-freed locations: all locations, including intermediate locations, that are reachable from ℓ_{First} are not freed. Importantly, this relation should satisfy the following properties:

1. M-reachability is pure, i.e. it is defined in the meta-logic, not as an Iris proposition.
2. It defines reachability between all locations in M including intermediate locations. For example,
$$\ell_{\mathsf{First}} \mapsto \mathtt{inj}_2\ (v, \mathsf{next}) * \mathsf{next} \mapsto \ell_{\mathsf{next}} \vdash \ulcorner \ell_{\mathsf{First}} \rightsquigarrow^{\mathsf{M}} \mathsf{next} \rightsquigarrow^{\mathsf{M}} \ell_{\mathsf{next}} \urcorner.$$
That is, the intermediate location next can both reach an be reached by the other locations.
3. Logical reachability should imply pure M-reachability:
$$\ell_n \overset{\gamma_{\mathsf{dom}},\gamma_{\mathsf{pt}}}{\rightsquigarrow} \ell_m \vdash \ulcorner \ell_n \rightsquigarrow^{\mathsf{M}} \ell_m \urcorner.$$

We define M-reachability inductively in Rocq using the following constructors:

Definition 8.5. (M-Reachability)

MReachRefl
$$\frac{\ell_n \in \mathsf{dom}(\mathsf{M})}{\ell_n \rightsquigarrow^{\mathsf{M}} \ell_n}$$

MReachStep
$$\frac{\mathsf{M}(\ell_n) = \mathtt{Some}(\mathtt{inj}_2\ ((ts, tid, v), \mathsf{next})) \qquad \mathsf{next} \rightsquigarrow^{\mathsf{M}} \ell_m}{\ell_n \rightsquigarrow^{\mathsf{M}} \ell_m}$$

MReachIntermediate
$$\frac{\mathsf{M}(\mathsf{next}) = \mathtt{Some}(\ell_{\mathsf{next}}) \qquad \ell_{\mathsf{next}} \rightsquigarrow^{\mathsf{M}} \ell_m}{\mathsf{next} \rightsquigarrow^{\mathsf{M}} \ell_m}$$

We can derive the following properties:

Lemma 8.6. (Properties of M-Reachability)

$$\frac{\ell_n \rightsquigarrow^{\mathsf{M}} \ell_o \qquad \ell_o \rightsquigarrow^{\mathsf{M}} \ell_m}{\ell_n \rightsquigarrow^{\mathsf{M}} \ell_m}\ \textsc{MReachTrans} \qquad \frac{\ell_n \rightsquigarrow^{\mathsf{M}} \ell_m \qquad \ell_m \rightsquigarrow^{\mathsf{M}} \ell_n \qquad \ell_n \rightsquigarrow^{\mathsf{M}} \ell_{\mathsf{dummy}} \qquad \mathsf{M}(\ell_{\mathsf{dummy}}) = \texttt{None}}{\ell_n = \ell_m}\ \textsc{MReachNoCycles}$$

Importantly, we can prove the desired implication by additionally owning the full view of the points-to map and its domain:

Lemma 8.7. (Reachability implies M-Reachability)

$$\frac{\ell_n \overset{\gamma_{\mathsf{dom}},\gamma_{\mathsf{pt}}}{\rightsquigarrow} \ell_m \qquad \boxed{\bullet \mathsf{M}}^{\gamma_{\mathsf{pt}}} \qquad \boxed{\bullet(\mathsf{dom}(\mathsf{M}))}^{\gamma_{\mathsf{dom}}}}{\ulcorner \ell_n \rightsquigarrow^{\mathsf{M}} \ell_m \urcorner}\ \textsc{ReachImplMReach}$$

To conclude; a location ℓ is not freed if it satisfies $\ell_{\mathsf{First}} \rightsquigarrow^{\mathsf{M}} \ell$.

8.2.2 FreeLater & To-Be-Freed

We give more ways for a location ℓ to be in the not-freed set.

In the dequeue program, we always free a location (unless the queue is empty). More precisely, we free either the first location ℓ_{First}, or the location ℓ_{FL} stored in FreeLater.
Additionally, in our HeapLang implementation, we also free the intermediate location, if it exists. That is, if

$$\ell_{\mathsf{First}} \mapsto \mathsf{inj}_2\ (v, \mathsf{next}) * \mathsf{next} \mapsto \ell_{\mathsf{next}}$$

for some value v and locations next and ℓ_{next}, then we free the points-tos $\ell_{\mathsf{First}} \mapsto \mathsf{inj}_2\ (v, \mathsf{next})$ and $\mathsf{next} \mapsto \ell_{\mathsf{next}}$.
Similarly for ℓ_{FL}; either it points to a dummy node, in which case there is no intermediate location to free, or we additionally free the location it points to.

We want to keep track of the locations that are going to be freed as precisely as possible. In particular, it is too strict to say that a location is not freed if and only if it is reachable from the current first location or stored in FreeLater. During dequeue, there are intermediate steps where locations are neither reachable from ℓ_{First} nor stored in FreeLater, but still have not yet been freed.

To capture this, we introduce a ghost resource containing a location tbf (to-be-freed). The type of tbf is inductively defined as:

$$\mathsf{tbfNone} \mid \mathsf{tbfCandidate}(\ell) \mid \mathsf{tbfFreeing}(\ell).$$

The default state is tbfNone. The state $\mathsf{tbfCandidate}(\ell)$ means that ℓ is a candidate to be freed, but has not yet been freed. The state $\mathsf{tbfFreeing}(\ell)$ means that ℓ is currently being freed. We only use this state when we want to free both the location ℓ and the intermediate location it points to. If tbf is in the state $\mathsf{tbfFreeing}(\ell)$, then we guarantee that the intermediate location pointed to by ℓ has not yet been freed.

To motivate how tbf is used, we consider the dequeue program more carefully:
Assume the the current first location is ℓ_{First}, and that $\ell_{\mathsf{First}} \mapsto \mathsf{inj}_2\ (v, \mathsf{next}) * \mathsf{next} \mapsto \ell_{\mathsf{next}}$ for some value v and locations next and ℓ_{next}. Furthermore, assume $\mathsf{FreeLater} \mapsto \ell_{\mathsf{FL}}$ for some location ℓ_{FL}.
When we store ℓ_{next} in First, the old first location, ℓ_{First}, becomes a candidate to be freed. At this point, we update tbf from tbfNone to $\mathsf{tbfCandidate}(\ell_{\mathsf{First}})$.

If ℓ_{First} is *not* the announced location, we update $\mathsf{tbfCandidate}(\ell_{\mathsf{First}})$ to $\mathsf{tbfFreeing}(\ell_{\mathsf{First}})$ and free ℓ_{First}. Finally, we free the intermediate location next and update $\mathsf{tbfFreeing}(\ell_{\mathsf{First}})$ to tbfNone.

If ℓ_{First} *is* the announced location, we cannot safely free it yet. Instead, we store ℓ_{First} in FreeLater, and update $\mathsf{tbfCandidate}(\ell_{\mathsf{First}})$ to $\mathsf{tbfCandidate}(\ell_{\mathsf{FL}})$. Intuitively, we can think of this as swapping the locations in tbf and FreeLater. Note that at the time of freeing ℓ_{FL}, the current value stored in FreeLater is ℓ_{First}, not ℓ_{FL}.

When we later free ℓ_{FL}, there are again two cases. If ℓ_{FL} points to an intermediate location, we update $\mathsf{tbfCandidate}(\ell_{\mathsf{FL}})$ to $\mathsf{tbfFreeing}(\ell_{\mathsf{FL}})$, free ℓ_{FL}, then free the intermediate location it points to, and finally update tbf to $\mathsf{tbfNone}$.
If ℓ_{FL} instead points to a dummy node, we update $\mathsf{tbfCandidate}(\ell_{\mathsf{FL}})$ to $\mathsf{tbfNone}$ directly when freeing ℓ_{FL}.

From these observations, we have five new properties that imply that a location has not been freed:

1. The current location stored in $\mathsf{FreeLater}$, ℓ_{FL}, is not freed, and
2. any intermediate location ℓ_{FL} points to is not freed.
3. The candidate to be freed, $\mathsf{tbfCandidate}(\ell_{\mathsf{tbf}})$ is not freed, and
4. the intermediate location ℓ_{tbf} points to is not freed.
5. The location being freed, $\mathsf{tbfFreeing}(\ell_{\mathsf{tbf}})$, *is* freed, but the intermediate location that ℓ_{tbf} points to is not freed.

That is, a location ℓ has not been freed if it satisfies:

$$\begin{aligned}&\ell = \ell_{\mathsf{FL}} \lor \big(\exists v, \mathsf{M}(\ell_{\mathsf{FL}}) = \mathtt{Some}(\mathsf{inj}_2\ (v, \ell))\big)\\ &\lor\ \mathsf{tbf} = \mathsf{tbfCandidate}(\ell) \lor \big(\exists v, \ell_{\mathsf{tbf}}.\ \mathsf{tbf} = \mathsf{tbfCandidate}(\ell_{\mathsf{tbf}}) \land \mathsf{M}(\ell_{\mathsf{tbf}}) = \mathtt{Some}(\mathsf{inj}_2\ (v, \ell))\big)\\ &\lor\ \big(\exists v, \ell_{\mathsf{tbf}}.\ \mathsf{tbf} = \mathsf{tbfFreeing}(\ell_{\mathsf{tbf}}) \land \mathsf{M}(\ell_{\mathsf{tbf}}) = \mathtt{Some}(\mathsf{inj}_2\ (v, \ell))\big).\end{aligned}$$

8.2.3 Defining the Not-Freed Set

Finally, we can define the set of locations that are not freed. Either, the location is M-reachable from the current first location (section 8.2.1), or it satisfies one of the properties described in section 8.2.2.

Definition 8.8. (Not-Freed Set)

$$\begin{aligned}&\mathsf{notFreedSetDef}(\mathsf{S}_{\mathsf{notFreed}}, \ell_{\mathsf{FL}}, \ell_{\mathsf{First}}, \mathsf{M}, \mathsf{tbf}) \triangleq\\ &\qquad \forall \ell,\ \ell \in \mathsf{S}_{\mathsf{notFreed}} \Leftrightarrow\\ &\qquad\quad \ell = \ell_{\mathsf{FL}} \lor (\exists v, \mathsf{M}(\ell_{\mathsf{FL}}) = \mathtt{Some}(\mathsf{inj}_2\ (v, \ell)))\\ &\qquad \lor\ \mathsf{tbf} = \mathsf{tbfCandidate}(\ell) \lor (\exists v, \ell_{\mathsf{tbf}}.\ \mathsf{tbf} = \mathsf{tbfCandidate}(\ell_{\mathsf{tbf}}) \land \mathsf{M}(\ell_{\mathsf{tbf}}) = \mathtt{Some}(\mathsf{inj}_2\ (v, \ell)))\\ &\qquad \lor\ (\exists v, \ell_{\mathsf{tbf}}.\ \mathsf{tbf} = \mathsf{tbfFreeing}(\ell_{\mathsf{tbf}}) \land \mathsf{M}(\ell_{\mathsf{tbf}}) = \mathtt{Some}(\mathsf{inj}_2\ (v, \ell)))\\ &\qquad \lor\ \ell_{\mathsf{First}} \rightsquigarrow^{\mathsf{M}} \ell.\end{aligned}$$

8.2.4 Defining logicalPts

We define the $\mathsf{logicalPts}$ predicate that we store in the invariant as follows:

Definition 8.9. (Logical Points-Tos)

$$\begin{aligned}&\mathsf{logicalPts}(\gamma_{\mathsf{pt}}, \gamma_{\mathsf{dom}}, \ell_{\mathsf{FL}}, \ell_{\mathsf{First}}, \mathsf{tbf}) \triangleq \exists \mathsf{M}, \mathsf{S}_{\mathsf{notFreed}}.\\ &\qquad \boxed{\bullet \mathsf{M}}^{\gamma_{\mathsf{pt}}} * \boxed{\bullet(\mathsf{dom}(\mathsf{M}))}^{\gamma_{\mathsf{dom}}} * \boxed{\circ(\mathsf{dom}(\mathsf{M}))}^{\gamma_{\mathsf{dom}}} *\\ &\qquad \ulcorner \mathsf{S}_{\mathsf{notFreed}} \subseteq \mathsf{dom}(\mathsf{M}) \urcorner * \ulcorner \mathsf{notFreedSetDef}(\mathsf{S}_{\mathsf{notFreed}}, \ell_{\mathsf{FL}}, \ell_{\mathsf{First}}, \mathsf{M}, \mathsf{tbf}) \urcorner *\\ &\qquad \mathop{\Large *}_{\ell \mapsto v^? \in \mathsf{M}} \mathsf{ptMapPts}(\mathsf{M}, \mathsf{S}_{\mathsf{notFreed}}, \ell, v^?).\end{aligned}$$

The logical points-tos that have not been freed depend on the current state of the queue; the current first location ℓ_{First}, the current location stored in $\mathsf{FreeLater}$, ℓ_{FL}, and the current to-be-freed location tbf. These are exactly the values used to define the set of not-freed locations $\mathsf{S}_{\mathsf{notFreed}}$.

We existentially quantify both the logical points-to map M and the set of not-freed locations $\mathsf{S}_{\mathsf{notFreed}}$. The set $\mathsf{S}_{\mathsf{notFreed}}$ is defined by the predicate

$$\mathsf{notFreedSetDef}(\mathsf{S}_{\mathsf{notFreed}}, \ell_{\mathsf{FL}}, \ell_{\mathsf{First}}, \mathsf{M}, \mathsf{tbf}),$$

which captures exactly the locations that have not yet been freed based on the observations from sections 8.2.1 and 8.2.2.

We keep the authoritative view of the points-to map M, together with the authoritative and fragmental views of its domain. We additionally maintain the invariant $\mathsf{S_{notFreed}} \subseteq \mathsf{dom}(\mathsf{M})$, which ensures that all locations we have assessed to not have been freed are in fact in the logical points-to map.

Finally, for every logical points-to stored in M, we keep either the corresponding real points-to or proof that the location has been freed, using ptMapPts (definition 8.2). Whether a real points-to is available depends on whether the location is in the set $\mathsf{S_{notFreed}}$: if $\ell \in \mathsf{S_{notFreed}}$, we keep the real points-to for ℓ; otherwise, we keep $\mathsf{freed}(\ell)$. Consequently, the set $\mathsf{S_{notFreed}}$ determines exactly which real points-tos are available in the invariant.

8.2.5 Changes to the Invariant

The main difference in this setting is the addition of the logical points-to map, logicalPts. Since we actually free locations, we can no longer use persistent points-tos to represent the linked list of the queue. Instead, the invariant stores logical points-tos, which can later be exchanged for real points-tos if we can prove that the location has not been freed. As a result, the invariant now additionally keeps track of which locations have and have not been freed. This leads to a few changes compared to the setting without freeing.

First, we replace the persistent points-tos that represent the queue by logical points-tos. More specifically, the points-tos that make up the linked list are replaced by either a ptPers or a ptNone predicate. For example, we redefine $\mathsf{isNodeList}(\ell_{\mathsf{First}}, \ell_{\mathsf{dummy}}, xs, \gamma_{\mathsf{pt}})$ as follows:

$$\begin{aligned}\mathsf{isNodeList}(\ell_{\mathsf{First}}, \ell_{\mathsf{dummy}}, xs, \gamma_{\mathsf{pt}}) \triangleq\ & \big(\ulcorner xs = \mathsf{nil}\urcorner * \ulcorner \ell_{\mathsf{First}} = \ell_{\mathsf{dummy}}\urcorner\big) \\ & \vee \\ & \big(\exists ts, tid, w.\ \ulcorner xs = (ts, tid, w) :: xs'\urcorner * \\ & \quad \exists \mathsf{next}, \ell.\ \ell_{\mathsf{First}} \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{\square} \mathsf{inj}_2\ ((ts, tid, w), \mathsf{next}) * \\ & \quad \mathsf{next} \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{\square} \ell * \mathsf{isNodeList}(\ell_{\mathsf{next}}, \ell_{\mathsf{dummy}}, xs', \gamma_{\mathsf{pt}})\big).\end{aligned}$$

Similarly, in the disjunction describing the last location, lastDisj, we use ptPers instead of persistent points-tos. For the points-to $\ell_{\mathsf{dummy}} \mapsto_{1/2} \mathsf{inj}_1\ ()$, we do not replace it with $\ell_{\mathsf{dummy}} \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{1/2} \mathsf{inj}_1\ ()$. Instead, we keep the half points-to in the invariant and give $\ell_{\mathsf{dummy}} \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{1/2} \mathsf{inj}_1\ ()$ to the enqueuer. We additionally include $\circ\{\ell_{\mathsf{dummy}}\}^{\gamma_{\mathsf{dom}}}$ as proof that the other half of the points-to is stored in the logical points-to map.

Another change is the addition of FreeLater to the invariant. As argued earlier, only the dequeuer reads from and writes to FreeLater. However, the invariant must also know its current value, since ℓ_{FL} is used in the definition of the not-freed set $\mathsf{S_{notFreed}}$. We therefore include half of the points-to

$$\mathsf{FreeLater} \mapsto_{1/2} \ell_{\mathsf{FL}}$$

in the invariant, together with the following predicate describing the location ℓ_{FL} stored in FreeLater:

$$\begin{aligned}&\mathsf{ptNoneOrPtPersReach}(\ell_{\mathsf{FL}}, \ell_{\mathsf{First}}, \gamma_{\mathsf{dom}}, \gamma_{\mathsf{pt}}) \triangleq \\ &\quad \ell_{\mathsf{FL}} \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{1/2} \mathsf{inj}_1\ () \vee \\ &\quad \big((\exists ts, tid, w, \mathsf{next}, \ell_{\mathsf{next}}.\ \ell_{\mathsf{FL}} \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{\square} \mathsf{inj}_2\ ((ts, tid, w), \mathsf{next}) * \mathsf{next} \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{\square} \ell_{\mathsf{next}}) \\ &\quad * \ell_{\mathsf{FL}} \overset{\gamma_{\mathsf{dom}}, \gamma_{\mathsf{pt}}}{\rightsquigarrow} \ell_{\mathsf{First}} * \ulcorner \ell_{\mathsf{FL}} \neq \ell_{\mathsf{First}}\urcorner\big).\end{aligned}$$

This predicate intuitively captures the two cases of the location ℓ_{FL} stored in FreeLater: either it points to a dummy node, or it points to a non-empty node and reaches the first location ℓ_{First} in at least one step ($\ell_{\mathsf{FL}} \neq \ell_{\mathsf{First}}$).

Additionally, we give the following supplementary information about ℓ_{FL} to the dequeuer:

$$\begin{aligned}\mathsf{isOptPts}(\ell_{\mathsf{FL}}, v^?_{\mathsf{FL}}) \triangleq\ & \big(\ulcorner v^?_{\mathsf{FL}} = \mathtt{None}\urcorner * \ell_{\mathsf{FL}} \mapsto_{1/2} \mathsf{inj}_1\ ()\big) \lor \\ & \big(\exists ts, tid, w, \mathsf{next}.\ \ulcorner v^?_{\mathsf{FL}} = \mathtt{Some}(ts, tid, w, \mathsf{next})\urcorner * \\ & \ell_{\mathsf{FL}} \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{\square} \mathsf{inj}_2\ ((ts, tid, w), \mathsf{next})\big).\end{aligned}$$

The definitions of $\mathsf{isOptPt}(\mathsf{Announce}, \ell^?_{\mathsf{A}})$ and $\mathsf{isOptVal}(\mathsf{Help}, v^?_{\mathsf{Help}})$ remain unchanged from the previous version (section 7.2.1).

Finally, we include the location to-be-freed tbf in the invariant. It captures the locations that are candidates to be freed or currently being freed. The invariant guarantees that tbf is either in the default state tbfNone, or that the location that is a candidate to be freed or currently being freed satisfies the predicate ptNoneOrPtPersReach. Additionally, we ensure that the location in tbf is never the same as the location stored in FreeLater.

$$\begin{aligned}\mathsf{tbfDisj}(\mathsf{tbf}, \ell_{\mathsf{First}}, \ell_{\mathsf{FL}}, \gamma_{\mathsf{dom}}, \gamma_{\mathsf{pt}}) \triangleq\ & \ulcorner \mathsf{tbf} = \mathsf{tbfNone}\urcorner \lor \\ & \big(\exists \ell_{\mathsf{tbf}},\ (\ulcorner \mathsf{tbf} = \mathsf{tbfCandidate}(\ell_{\mathsf{tbf}})\urcorner \lor \ulcorner \mathsf{tbf} = \mathsf{tbfFreeing}(\ell_{\mathsf{tbf}})\urcorner) * \\ & \mathsf{ptNoneOrPtPersReach}(\ell_{\mathsf{tbf}}, \ell_{\mathsf{First}}, \gamma_{\mathsf{dom}}, \gamma_{\mathsf{pt}}) * \ulcorner \ell_{\mathsf{tbf}} \neq \ell_{\mathsf{FL}}\urcorner\big).\end{aligned}$$

8.3 logicalPts Lemmas

Instead of going through the proofs of the specifications of the individual operations, we focus on the lemmas for updating the logical points-to map logicalPts. These lemmas capture the necessary reasoning about freeing and accessing points-tos in the algorithm that we must do in the proofs.

The main challenge is to maintain the set of not-freed locations $\mathsf{S}_{\mathsf{notFreed}}$ according to its definition (section 8.2.3): Every time we add a logical points-to to the points-to map M, we have to show that the added location satisfies at least one of the conditions for being in $\mathsf{S}_{\mathsf{notFreed}}$. Similarly, whenever we free a points-to, we have to prove that the location no longer satisfies any of these conditions. This requires reasoning about e.g. how the M-reachability changes when M is modified, as well as reasoning about the current values of tbf, ℓ_{FL} and ℓ_{First}, and how they relate to each other.
These small details complicate the proofs of the lemmas. For example, we need many assumptions in each lemma to rule out impossible cases; the shape of ℓ_{FL}, that the points-to of the dummy node ℓ_{dummy} is in fact in the points-to map, that the tbf satisfies certain properties, and so on. Since all proofs are mechanised in Rocq, we will spare the reader some of the details.

We show the lemmas here, since the points-to map and the logical points-tos play a significant role in the proofs of the specifications in this setting where we explicitly free memory. The main point is that we have to know exactly when a points-to in the queue has or has not yet been freed. That is why we are so specific about how ℓ_{FL}, ℓ_{First} and tbf relate to each other; their relationships define exactly which locations are and are not safe to free.

Initialisation

AllocLogicalPts

$$\frac{\boxed{\bullet\emptyset}^{\gamma_{\mathsf{pt}}} \qquad \boxed{\bullet(\mathsf{dom}(\emptyset))}^{\gamma_{\mathsf{dom}}} \qquad \ell_{\mathsf{FL}} \mapsto \mathsf{inj}_1\ ()}{\Rrightarrow \Big(\mathsf{logicalPts}(\gamma_{\mathsf{pt}}, \gamma_{\mathsf{dom}}, \ell_{\mathsf{FL}}, \ell_{\mathsf{First}}, \mathsf{tbfNone}) * \ell_{\mathsf{FL}} \mapsto_{1/2} \mathsf{inj}_1\ () * \ell_{\mathsf{FL}} \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{1/2} \mathsf{inj}_1\ ()\Big)}$$

AddFirstNoneToM

$$\frac{\mathsf{logicalPts}(\gamma_{\mathsf{pt}}, \gamma_{\mathsf{dom}}, \ell_{\mathsf{FL}}, \ell_{\mathsf{First}}, \mathsf{tbfNone}) \qquad \ell_{\mathsf{First}} \mapsto \mathsf{inj}_1\ ()}{\Rrightarrow \Big(\mathsf{logicalPts}(\gamma_{\mathsf{pt}}, \gamma_{\mathsf{dom}}, \ell_{\mathsf{FL}}, \ell_{\mathsf{First}}, \mathsf{tbfNone}) * \ell_{\mathsf{First}} \mapsto_{1/2} \mathsf{inj}_1\ () * \ell_{\mathsf{First}} \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{1/2} \mathsf{inj}_1\ ()\Big)}$$

These two initialisation lemmas of logicalPts are used in the queue intitialisation to allocate the points-to map and add the two first points-tos; $\ell_{\mathsf{FL}} \mapsto \mathsf{inj}_1\ ()$ and $\ell_{\mathsf{First}} \mapsto \mathsf{inj}_1\ ()$.

The lemma AllocLogicalPts allows us to initialise the collection of logical points-tos logicalPts. First, we allocate the empty points-to map M and the empty domain. Initially, we add half of the FreeLater points-to, $\ell_{\mathsf{FL}} \mapsto \mathsf{inj}_1\ ()$ to the points-to map. We get half of it back, and a ptNone for the half that we put in the map.
The next lemma, AddFirstNoneToM, allows us to add $\ell_{\mathsf{First}} \mapsto_{1/2} \mathsf{inj}_1\ ()$ to the points-to map, and similarly, we get the other half $\ell_{\mathsf{First}} \mapsto_{1/2} \mathsf{inj}_1\ ()$ and a ptNone.

Reading $\ell \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{\square} v$ (readFront & dequeue)

ReadPtPers

$$\frac{\mathsf{logicalPts}(\gamma_{\mathsf{pt}}, \gamma_{\mathsf{dom}}, \ell_{\mathsf{FL}}, \ell_{\mathsf{First}}, \mathsf{tbf}) \qquad \ell \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{\square} v \qquad (\ell_{\mathsf{First}} \overset{\gamma_{\mathsf{dom}}, \gamma_{\mathsf{pt}}}{\rightsquigarrow} \ell \ \vee\ \ulcorner \mathsf{tbf} = \mathsf{tbfCandidate}(\ell) \urcorner \ \vee\ \ulcorner \ell_{\mathsf{FL}} = \ell \urcorner)}{\ell \mapsto v * (\ell \mapsto v \mathrel{-\!\!*} \mathsf{logicalPts}(\gamma_{\mathsf{pt}}, \gamma_{\mathsf{dom}}, \ell_{\mathsf{FL}}, \ell_{\mathsf{First}}, \mathsf{tbf}))}$$

This lemma is used when either the dequeuer or enqueuer wants to read the value of a ptPers, $\ell \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{\square} v$. In particular, the enqueuer applies this lemma in readFrontEnq to read points-to of the announced first location ℓ_{FL}.

As mentioned earlier, exchanging a logical points-to for a real points-to requires proving that the location has not been freed. By the definition of the set $\mathsf{S}_{\mathsf{notFreed}}$ (section 8.2.3), it suffices to prove that the location is reachable from the current first location, is the candidate to be freed, or is stored in FreeLater.

Lemma for enqueue
In the enqueue operation, we only update the logicalPts once; namely when we update the dummy node ℓ to store the enqueued value and the pointer to the new dummy node ℓ_{dummy}, i.e. store $\mathsf{inj}_2\ ((ts, tid, w), \mathsf{next})$ in ℓ, where $\mathsf{next} \mapsto \ell_{\mathsf{dummy}}$. Before this step, the enqueuer has allocated a new dummy node $\ell_{\mathsf{dummy}} \mapsto \mathsf{inj}_1\ ()$ and a pointer to it, $\mathsf{next} \mapsto \ell_{\mathsf{dummy}}$.
The invariant owns half of the normal points-to for the dummy node, $\ell \mapsto_{1/2} \mathsf{inj}_1\ ()$, and the enqueuer always owns $\ell \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{1/2} \mathsf{inj}_1\ ()$. This indicates that the other half of the real points-to is stored in the points-to map.

EnqueueAddPts

$$\frac{\begin{array}{c}\mathsf{logicalPts}(\gamma_{\mathsf{pt}}, \gamma_{\mathsf{dom}}, \ell_{\mathsf{FL}}, \ell_{\mathsf{First}}, \mathsf{tbf}) \qquad \ell \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{1/2} \mathsf{inj}_1\ () \qquad \ell \mapsto_{1/2} \mathsf{inj}_1\ () \qquad \ell_{\mathsf{First}} \overset{\gamma_{\mathsf{dom}}, \gamma_{\mathsf{pt}}}{\rightsquigarrow} \ell \qquad \ulcorner \ell_{\mathsf{FL}} \neq \ell \urcorner \\ (\ulcorner \mathsf{tbf} = \mathsf{tbfNone} \urcorner \vee (\exists \ell_{\mathsf{tbf}},\ (\ulcorner \mathsf{tbf} = \mathsf{tbfCandidate}(\ell_{\mathsf{tbf}}) \urcorner \vee \ulcorner \mathsf{tbf} = \mathsf{tbfFreeing}(\ell_{\mathsf{tbf}}) \urcorner) * \ulcorner \ell_{\mathsf{tbf}} \neq \ell \urcorner))\end{array}}{\begin{array}{c}\ell \mapsto \mathsf{inj}_1\ () * \Big(\ell \mapsto \mathsf{inj}_2\ ((ts, tid, w), \mathsf{next}) * \mathsf{next} \mapsto \ell_{\mathsf{dummy}} * \ell_{\mathsf{dummy}} \mapsto \mathsf{inj}_1\ () \mathrel{-\!\!*} \Rightarrow \\ \mathsf{logicalPts}(\gamma_{\mathsf{pt}}, \gamma_{\mathsf{dom}}, \ell_{\mathsf{FL}}, \ell_{\mathsf{First}}, \mathsf{tbf}) * \ell \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{\square} \mathsf{inj}_2\ ((ts, tid, w), \mathsf{next}) * \\ \mathsf{next} \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{\square} \ell_{\mathsf{dummy}} * \ell_{\mathsf{dummy}} \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{1/2} \mathsf{inj}_1\ () * \ell_{\mathsf{dummy}} \mapsto_{1/2} \mathsf{inj}_1\ ()\Big)\end{array}}$$

We first exchange $\ell \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{1/2} \mathsf{inj}_1\ ()$ for the real points-to $\ell \mapsto_{1/2} \mathsf{inj}_1\ ()$. This allows us to combine the two halves of the points-to, $\ell \mapsto \mathsf{inj}_1\ ()$, such that we can store the updated node value $\mathsf{inj}_2\ ((ts, tid, w), \mathsf{next})$ in ℓ. To do this, it must exchange $\ell \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{1/2} \mathsf{inj}_1\ ()$ for $\ell \mapsto_{1/2} \mathsf{inj}_1\ ()$ by proving that ℓ is in the set of not-freed locations. This is easy, since the first location always reaches the last location: $\ell_{\mathsf{First}} \overset{\gamma_{\mathsf{dom}}, \gamma_{\mathsf{pt}}}{\rightsquigarrow} \ell$.

After performing the store, we own the points-to $\ell \mapsto \mathsf{inj}_2\ ((ts, tid, w), \mathsf{next})$. Since the points-to map should always store all points-tos that make up the linked list representing the queue, the next step is to add this points-to to the points-to map, as well as the points-tos $\mathsf{next} \mapsto \ell_{\mathsf{dummy}}$ and $\ell_{\mathsf{dummy}} \mapsto \mathsf{inj}_1\ ()$.
We exchange $\ell \mapsto \mathsf{inj}_2\ ((ts, tid, w), \mathsf{next})$ and $\mathsf{next} \mapsto \ell_{\mathsf{dummy}}$ for their corresponding ptPers logical points-tos, and we exchange $\ell_{\mathsf{dummy}} \mapsto \mathsf{inj}_1\ ()$ for $\ell_{\mathsf{dummy}} \mapsto_{1/2} \mathsf{inj}_1\ ()$ and its corresponding ptNone.

We need the rest of the assumptions, e.g. $\ulcorner \ell_{\mathsf{FL}} \neq \ell \urcorner$ and the disjunction about all the possibilities for tbf, to help us maintain the not-freed set for the newly added points-tos in the points-to map.

Lemmas for dequeue
In the dequeue operation, we make several updates to the points-to map.

First, the dequeuer moves the First pointer from ℓ_{First} to point to the next location ℓ_{next}:

DequeueMoveFirst

$$\frac{\mathsf{logicalPts}(\gamma_{\mathsf{pt}}, \gamma_{\mathsf{dom}}, \ell_{\mathsf{FL}}, \ell_{\mathsf{First}}, \mathsf{tbfNone}) \qquad \ell_{\mathsf{First}} \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{\square} \mathsf{inj}_2\,((ts, tid, w), \mathsf{next}) \qquad \mathsf{next} \mapsto \ell_{\mathsf{next}} \qquad \boxed{\circ\{\mathsf{next}\}}^{\gamma_{\mathsf{dom}}}}{\Rightarrow\!\!\!\!\mid\, \mathsf{logicalPts}(\gamma_{\mathsf{pt}}, \gamma_{\mathsf{dom}}, \ell_{\mathsf{FL}}, \ell_{\mathsf{next}}, \mathsf{tbfCandidate}(\ell_{\mathsf{First}}))}$$

This is relatively simple; we just need to know that the current first location ℓ_{First} actually points to the next location ℓ_{next}.
Note that we update tbf from tbfNone to $\mathsf{tbfCandidate}(\ell_{\mathsf{First}})$, since ℓ_{First} now becomes a candidate to be freed (see section 8.2.2). This ensures that ℓ_{First} is still in the not-freed set, even though it is no longer reachable from the first location as before.

If the dequeued location ℓ_{First} is the announced location, we swap the location in tbf, $\mathsf{tbfCandidate}(\ell_{\mathsf{First}})$, and the location stored in FreeLater, ℓ_{FL}:

DequeueSwapFreeLaterAndTbf

$$\frac{\mathsf{logicalPts}(\gamma_{\mathsf{pt}}, \gamma_{\mathsf{dom}}, \ell_{\mathsf{FL}}, \ell_{\mathsf{next}}, \mathsf{tbfCandidate}(\ell_{\mathsf{First}}))}{\Rightarrow\!\!\!\!\mid\, \mathsf{logicalPts}(\gamma_{\mathsf{pt}}, \gamma_{\mathsf{dom}}, \ell_{\mathsf{First}}, \ell_{\mathsf{next}}, \mathsf{tbfCandidate}(\ell_{\mathsf{FL}}))}$$

It is a simple lemma with a straightforward proof, since we just swap the roles of ℓ_{First} and ℓ_{FL}; this preserves the not-freed set $\mathsf{S}_{\mathsf{notFreed}}$ and the points-to map M.

We have five lemmas concerning freeing locations. In section 8.2.2, we have explained the different cases, depending on whether we free ℓ_{FL} or ℓ_{First}, and whether ℓ_{FL} points to a dummy node.
These lemmas are complicated because we have to make sure that the freed location is no longer in the not-freed set, such that no thread can ever try to load a freed location. We need many assumptions to rule out edge cases. We will sketch the most important takeaways from selected lemmas.

We show the two lemmas for freeing the previous first location ℓ_{First} and its intermediate location next. At this point in the dequeue program, the current first location is ℓ_{next}, and the location in FreeLater is ℓ_{FL}. Furthermore, we have already updated tbf to $\mathsf{tbfCandidate}(\ell_{\mathsf{First}})$.

DequeueFreeFirst

$$\frac{\begin{array}{c}\mathsf{logicalPts}(\gamma_{\mathsf{pt}}, \gamma_{\mathsf{dom}}, \ell_{\mathsf{FL}}, \ell_{\mathsf{next}}, \mathsf{tbfCandidate}(\ell_{\mathsf{First}})) \\ \ell_{\mathsf{First}} \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{\square} \mathsf{inj}_2\,((ts, tid, w), \mathsf{next}) \qquad \mathsf{next} \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{\square} \ell_{\mathsf{next}} \qquad \ulcorner \ell_{\mathsf{First}} \neq \ell_{\mathsf{FL}} \urcorner \qquad \ulcorner \ell_{\mathsf{First}} \neq \ell_{\mathsf{next}} \urcorner \\ \ell_{\mathsf{dummy}} \mapsto_{1/2} \mathsf{inj}_1\,() \qquad \boxed{\circ\{\ell_{\mathsf{dummy}}\}}^{\gamma_{\mathsf{dom}}} \qquad \ell_{\mathsf{next}} \overset{\gamma_{\mathsf{dom}}, \gamma_{\mathsf{pt}}}{\rightsquigarrow} \ell_{\mathsf{dummy}} \\ \mathsf{ptNoneOrPtPersReach}(\ell_{\mathsf{FL}}, \ell_{\mathsf{next}}, \gamma_{\mathsf{dom}}, \gamma_{\mathsf{pt}})\end{array}}{\begin{array}{c}(\exists v,\ \ell_{\mathsf{First}} \mapsto v) \ *\ \Big(\mathsf{freed}(\ell_{\mathsf{First}}) \mathrel{-\!\!*} \Rightarrow\!\!\!\!\mid \\ \mathsf{logicalPts}(\gamma_{\mathsf{pt}}, \gamma_{\mathsf{dom}}, \ell_{\mathsf{FL}}, \ell_{\mathsf{next}}, \mathsf{tbfFreeing}(\ell_{\mathsf{First}})) * \ell_{\mathsf{dummy}} \mapsto_{1/2} \mathsf{inj}_1\,() * \mathsf{ptNoneOrPtPersReach}(\ell_{\mathsf{FL}}, \ell_{\mathsf{next}}, \gamma_{\mathsf{dom}}, \gamma_{\mathsf{pt}})\Big)\end{array}}$$

In DequeueFreeFirst, we exchange the persistent logical points-to assertion $\ell_{\mathsf{First}} \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{\square} \mathsf{inj}_2\,((ts, tid, w), \mathsf{next})$ for a real points-to $\ell_{\mathsf{First}} \mapsto v$. We do not care that the values match, since we are just going to free it. After freeing, we have to give back $\mathsf{freed}(\ell_{\mathsf{First}})$ to the points-to map. We update tbf from $\mathsf{tbfCandidate}(\ell_{\mathsf{First}})$ to $\mathsf{tbfFreeing}(\ell_{\mathsf{First}})$, indicating that we have freed ℓ_{First} and will free the intermediate location next it points to (see section 8.2.2). In the proof of this lemma, we have to prove that ℓ_{First} does not satisfy the properties for not being freed (definition 8.8).

DequeueFreeFirstNext

$$\frac{\begin{array}{c}\mathsf{logicalPts}(\gamma_{\mathsf{pt}}, \gamma_{\mathsf{dom}}, \ell_{\mathsf{FL}}, \ell_{\mathsf{next}}, \mathsf{tbfFreeing}(\ell_{\mathsf{First}})) \\ \ell_{\mathsf{First}} \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{\square} \mathsf{inj}_2\,((ts, tid, w), \mathsf{next}) \qquad \mathsf{next} \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{\square} \ell_{\mathsf{next}} \\ \ell_{\mathsf{dummy}} \mapsto_{1/2} \mathsf{inj}_1\,() \qquad \boxed{\circ\{\ell_{\mathsf{dummy}}\}}^{\gamma_{\mathsf{dom}}} \qquad \ell_{\mathsf{next}} \overset{\gamma_{\mathsf{dom}}, \gamma_{\mathsf{pt}}}{\rightsquigarrow} \ell_{\mathsf{dummy}} \\ \mathsf{isOptPts}(\ell_{\mathsf{FL}}, v^?_{\mathsf{FL}}, \gamma_{\mathsf{pt}}) \qquad \Big(\ulcorner v^?_{\mathsf{FL}} = \mathtt{None} \urcorner \vee (\ell_{\mathsf{FL}} \overset{\gamma_{\mathsf{dom}}, \gamma_{\mathsf{pt}}}{\rightsquigarrow} \ell_{\mathsf{First}} * \ulcorner \mathsf{isSome}(v^?_{\mathsf{FL}}) \urcorner)\Big) \\ \ulcorner \ell_{\mathsf{First}} \neq \ell_{\mathsf{FL}} \urcorner \qquad \ulcorner \ell_{\mathsf{First}} \neq \ell_{\mathsf{next}} \urcorner \qquad \mathsf{ptNoneOrPtPersReach}(\ell_{\mathsf{FL}}, \ell_{\mathsf{next}}, \gamma_{\mathsf{dom}}, \gamma_{\mathsf{pt}})\end{array}}{\begin{array}{c}(\exists v,\ \mathsf{next} \mapsto v) \ *\ \Big(\mathsf{freed}(\mathsf{next}) \mathrel{-\!\!*} \Rightarrow\!\!\!\!\mid \mathsf{logicalPts}(\gamma_{\mathsf{pt}}, \gamma_{\mathsf{dom}}, \ell_{\mathsf{FL}}, \ell_{\mathsf{next}}, \mathsf{tbfNone}) * \ell_{\mathsf{dummy}} \mapsto_{1/2} \mathsf{inj}_1\,() * \\ \mathsf{ptNoneOrPtPersReach}(\ell_{\mathsf{FL}}, \ell_{\mathsf{next}}, \gamma_{\mathsf{dom}}, \gamma_{\mathsf{pt}}) * \mathsf{isOptPts}(\ell_{\mathsf{FL}}, v^?_{\mathsf{FL}}, \gamma_{\mathsf{pt}})\Big)\end{array}}$$

We free next using lemma DequeueFreeFirstNext; here, we similarly exchange a persistent logical points-to for a real points-to. For this, we prove that $\mathsf{next} \in \mathsf{S}_{\mathsf{notFreed}}$ since $\mathsf{tbf} = \mathsf{tbfFreeing}(\ell_{\mathsf{First}}) \wedge \mathsf{M}(\ell_{\mathsf{First}}) = \mathtt{Some}(\mathtt{inj}_2\ (v, \mathsf{next}))$ (see section 8.2.3). After freeing next, we put $\mathsf{freed}(\mathsf{next})$ back into the points-to map. To ensure that next is not in the not-freed set, we update $\mathsf{tbfFreeing}(\ell_{\mathsf{First}})$ to tbfNone.

We also have the lemmas for freeing ℓ_{FL}, the value stored in FreeLater, and possibly the intermediate location it points to, if it exists. These lemmas are very similar to freeing ℓ_{First} and next. They are available in appendix A.1 (and of course in the Rocq mechanisation).

8.4 Announce

The most complicated aspect of the concurrent proof is arguably the communication between the enqueuer and the dequeuer about the *announced* location.[4] The purpose of the announcement is to guarantee that the dequeuer never frees a node that the enqueuer will read.

Recall the readFrontEnq program: the enqueuer first reads the current first location ℓ_{First} stored in First, stores this location in Announce, and then checks whether the first location has changed. If the first location is unchanged, the enqueuer reads the value directly from ℓ_{First}. Otherwise, it reads the value stored in Help. Between all of these steps, the dequeuer may have modified the queue.

The goal is to capture why it is safe for the enqueuer to read from ℓ_{First} if the first pointer did not change. Intuitively, if the enqueuer announces a location ℓ_{First} and the dequeuer – who is concurrently dequeueing ℓ_{First} – observes the announcement, then the dequeuer will not free ℓ_{First}. This is part of the checks in the dequeue program.

To reason about this behaviour, we introduce a ghost variable $\ell^?_{\mathsf{fstAtAnno}}$ that stores the value of First *at the time of announcement*. This allows us to distinguish between two situations:

1. The announced location ℓ_{First} was still the first location at the time of announcement (i.e. when the enqueuer performs $\mathsf{Announce} \leftarrow \ell_{\mathsf{First}}$), in which case ℓ_{First} may still be safe to read.
2. The first location was changed *before* the enqueuer's announcement of ℓ_{First}. In this case, ℓ_{First} might not be safe to read. In the readFrontEnq program, it means that the enqueuer reads the value stored in Help instead of ℓ_{First}.

The invariant therefore contains an implication that captures this. We define:

$$\mathsf{fstAtAnnoAnnoImpl}(\ell^?_{\mathsf{fstAtAnno}}, \ell^?_{\mathsf{A}}, \ell_{\mathsf{First}}, \mathsf{tbf}, \ell_{\mathsf{FL}}, \gamma_{\mathsf{dom}}, \gamma_{\mathsf{pt}}) \triangleq$$
$$\forall \ell_{\mathsf{A}}, \left(\ulcorner \ell^?_{\mathsf{A}} = \mathtt{Some}(\ell_{\mathsf{A}})\urcorner * \ulcorner \ell^?_{\mathsf{fstAtAnno}} = \mathtt{Some}(\ell_{\mathsf{A}})\urcorner\right)$$
$$-\!\!* \ (\ell_{\mathsf{First}} \overset{\gamma_{\mathsf{dom}}, \gamma_{\mathsf{pt}}}{\rightsquigarrow} \ell_{\mathsf{A}} \vee \ulcorner \mathsf{tbf} = \mathsf{tbfCandidate}(\ell_{\mathsf{A}})\urcorner \vee \ulcorner \ell_{\mathsf{FL}} = \ell_{\mathsf{A}}\urcorner).$$

If the announced location ℓ_{A} coincides with the first location at the time of announcement, $\ell^?_{\mathsf{fstAtAnno}}$, then the announced location must either still be reachable from the current first location (i.e. $\ell_{\mathsf{First}} \overset{\gamma_{\mathsf{dom}}, \gamma_{\mathsf{pt}}}{\rightsquigarrow} \ell_{\mathsf{A}}$), be the current candidate to-be-freed ($\mathsf{tbf} = \mathsf{tbfCandidate}(\ell_{\mathsf{A}})$), or be is stored in FreeLater ($\ell_{\mathsf{A}} = \ell_{\mathsf{FL}}$). These properties correspond precisely to cases in which the location ℓ_{A} is known not to have been freed (see section 8.2.3). Hence, the enqueuer can use this implication to safely read from the logical persistent points-to of the announced location using lemma ReadPtPers.

To reason more precisely about the announced location, we add a ghost state that remembers all announced locations, $\mathsf{S}_{pastAnnos}$. We store the authoritative part of the set of all past (and current) announced locations, $\boxed{\bullet \mathsf{S}_{pastAnnos}}^{\gamma_{\mathsf{pastAnnos}}}$, in the invariant. The RA used for the set of past announcements is $Auth(gset(Loc))$. It satisfies, among other things, that

$$\boxed{\circ \{\ell_{\mathsf{A}}\}}^{\gamma_{\mathsf{pastAnnos}}} * \boxed{\bullet \mathsf{S}_{pastAnnos}}^{\gamma_{\mathsf{pastAnnos}}} \vdash \ulcorner \ell_{\mathsf{A}} \in \mathsf{S}_{pastAnnos}\urcorner.$$

[4] The interested reader is referred to the proof of the specification for readFrontEnq in Rocq.

We define the disjunction for the two cases of $\mathsf{S}_{pastAnnos}$ as follows:

$$\begin{aligned}&\mathsf{pastAnnosDisj}(\mathsf{S}_{pastAnnos}, \ell^?_{\mathsf{A}}, \ell_{\mathsf{First}}, \gamma_{\mathsf{pastAnnos}}, \gamma_{\mathsf{dom}}, \gamma_{\mathsf{pt}}) \triangleq \\ &\quad \left(\ulcorner \mathsf{S}_{pastAnnos} = \emptyset \urcorner * \ulcorner \ell^?_{\mathsf{A}} = \texttt{None} \urcorner\right) \lor \\ &\quad \left(\exists \ell_{\mathsf{A}}, \ulcorner \ell^?_{\mathsf{A}} = \texttt{Some}(\ell_{\mathsf{A}}) \urcorner * \mathsf{pastAnnosReach}(\ell_{\mathsf{A}}, \mathsf{S}_{pastAnnos}, \gamma_{\mathsf{dom}}, \gamma_{\mathsf{pt}}) * \ell_{\mathsf{A}} \overset{\gamma_{\mathsf{dom}}, \gamma_{\mathsf{pt}}}{\rightsquigarrow} \ell_{\mathsf{First}} * \boxed{\circ\{\ell_{\mathsf{A}}\}}^{\gamma_{\mathsf{pastAnnos}}}\right),\end{aligned}$$

where

$$\mathsf{pastAnnosReach}(\ell_{\mathsf{A}}, \mathsf{S}_{pastAnnos}, \gamma_{\mathsf{dom}}, \gamma_{\mathsf{pt}}) \triangleq \forall \ell, \ulcorner \ell \in \mathsf{S}_{pastAnnos} \urcorner \mathrel{-\!\!*} \ell \overset{\gamma_{\mathsf{dom}}, \gamma_{\mathsf{pt}}}{\rightsquigarrow} \ell_{\mathsf{A}}.$$

The purpose of this set is similar to the abstract reachability; we want to keep the information that all past announced location can reach the current announced locations. This reachability argument allows us to rule out some impossible cases in the proof.

With these additions to the invariant, the enqueuer can use the invariant to safely read the announced location in the readFrontEnq program. The dequeuer, on the other hand, has to maintain the invariant at every step of the dequeue program, including moving the first pointer, swapping the locations in FreeLater and tbf, and freeing locations. All of these changes and updates have to maintain the guarantee that the currently announced location has not been freed.

Therefore, the invariant stores additional auxiliary ghost state. In particular, we introduce a location $\ell_{\mathsf{not\text{-}fstAtAnno}}$, which allows the dequeuer to record a location that is known *not* to be the first location at the time of announcement. Similarly, we introduce an exception-pair $(\ell^?_{\mathsf{X\text{-}A}}, \ell^?_{\mathsf{X\text{-}fstAtAnno}})$ which the dequeuer can use to satisfy the invariant when it does not need to make guarantees about the currently announced location and the current first-location-at-the-time-of-announcement.

The purpose of these auxiliary variables is to allow the dequeuer to preserve enough information about earlier announcements, First and FreeLater locations, to rule out impossible cases when closing the invariant. We give the precise definitions for these in appendix A.2.

The main point is that the dequeuer must always be able to prove that the location it frees *cannot* coincide with the currently announced location. The multiple proofs of this fact in all different cases rely on the reachability relation between the involved locations in the queue (e.g. ReachNoCycles), as well as the auxiliary ghost state introduced above.

Overall, the announced location and the communication around it via the invariant plays a significant part in the proofs of the specification in the concurrent setting with freeing. Consequently, we need additional ghost resources to keep track of the different possible states of the queue with respect to the the announced location.

8.5 Specification

The specification in the concurrent setting with freeing is almost identical to the specification we gave in the concurrent setting without freeing (section 7.3). We give the same guarantees for the dequeued values, i.e. that they satisfy $\Phi(w)$, and require that the enqueuer can provide $\Phi(w)$ for all enqueued values.
The only differences are the use of the isQueueConcFreeing predicate instead of isQueueConc, and that we extend isEnqueuer and isDequeuer to include the additional ghost resources needed to reason about

freeing and announcements.

$$\begin{aligned}\mathsf{isEnqueuer}(G, \mathsf{Last}, \mathsf{Announce}) \triangleq\ & \exists \ell_{\mathsf{Last}}, \ell^?_{\mathsf{A}}, \ell^?_{\mathsf{fstAtAnno}}.\\ & \mathsf{Last} \mapsto_{1/2} \ell_{\mathsf{Last}} *\\ & \mathsf{isOptPt}(\mathsf{Announce}, \ell^?_{\mathsf{A}}) *\\ & \ell_{\mathsf{Last}} \overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{1/2} \mathsf{inj}_1\ () *\\ & \boxed{\tfrac{1}{2}(\ell^?_{\mathsf{fstAtAnno}})}^{\gamma_{\mathsf{FstAtAnno}}}.\end{aligned}$$

$$\begin{aligned}\mathsf{isDequeuer}(G, \mathsf{First}, \mathsf{FreeLater}, \mathsf{Help}) \triangleq\ & \exists \ell_{\mathsf{First}}, \ell_{\mathsf{FL}}, v^?_{\mathsf{Help}}, v^?_{\mathsf{FL}}, \ell^?_{\mathsf{X\text{-}A}}, \ell^?_{\mathsf{X\text{-}fstAtAnno}}, \ell_{\mathsf{not\text{-}fstAtAnno}}.\\ & \mathsf{First} \mapsto_{1/2} \ell_{\mathsf{First}} *\\ & \mathsf{FreeLater} \mapsto_{1/2} \ell_{\mathsf{FL}} *\\ & \mathsf{isOptVal}(\mathsf{Help}, v^?_{\mathsf{Help}}) *\\ & \mathsf{isOptPts}(\ell_{\mathsf{FL}}, v^?_{\mathsf{FL}}) *\\ & \boxed{\tfrac{1}{2}(\mathsf{tbfNone})}^{\gamma_{\mathsf{tbf}}} *\\ & \boxed{\tfrac{1}{2}(\ell^?_{\mathsf{X\text{-}A}}, \ell^?_{\mathsf{X\text{-}fstAtAnno}})}^{\gamma_{\mathsf{X}}} *\\ & \boxed{\tfrac{1}{2}(\ell_{\mathsf{not\text{-}fstAtAnno}})}^{\gamma_{\mathsf{not\text{-}fstAtAnno}}}.\end{aligned}$$

9 Conclusion and Future Work

In this report, we have verified the SESD wait-free queue algorithm [JP05] in Iris. We first proved a sequential specification of the queue. We then considered a simplified version of the SESD queue algorithm without explicit freeing of memory in a concurrent setting. The main challenge in this setting was to define a persistent queue predicate with an appropriate invariant to allow the enqueuer and dequeuer to work on the queue concurrently. The concurrent specification ensures that every dequeued value w satisfies a predicate Φ, and the enqueuer must prove that each enqueued value satisfies $\Phi(w)$. However, the specification gives no guarantees about the FIFO queue behaviour. Finally, we proved the concurrent queue specification for the full SESD algorithm including freeing. This required a more advanced invariant; we included the logicalPts predicate for a map of logical points tos which tracks exactly which locations have been freed. Furthermore, we added additional ghost states regarding the announced location, to ensure that the dequeuer never frees a location that the enqueuer may still read.

There are several possibilities for future work.

* **Higher-order concurrent abstract predicates (HOCAP) specification**
  The concurrent specification presented in this project is relatively weak in the sense that an implementation that always returns `None` for both dequeue and readFront would still satisfy the specification. Furthermore, the specification does not guarantee that the data structure behaves as a FIFO queue, as opposed to e.g. a stack or a concurrent bag.

  A stronger specification for the SESD queue is a *higher-order concurrent abstract predicates* (HOCAP) specification [SBP13]. HOCAP specifications provide a more abstract and modular interface to concurrent data structures, where clients interact with an abstract predicate describing the logical behaviour of the data structure.

  The main idea would be to introduce additional ghost state representing the abstract contents of the queue, for example as a mathematical list of queue elements. The queue operations would then update this ghost state according to the intended FIFO behaviour of the queue. This would make it possible to prove stronger guarantees, such as that the dequeued values correspond to previously enqueued values and are returned in FIFO order.

  A HOCAP-style specification for the Michael-Scott queue was given in [Ped24]. The SESD queue and the Michael-Scott queue are structurally similar, and since we already use related techniques such as reachability arguments in our verification of the concurrent specification, which suggests that we might be able to prove a HOCAP specification for the SESD queue in a similar way to the Michael-Scott queue.

* **Verification of the full MESD algorithm**
  A natural next step is to verify the full multiple enqueuer, single dequeuer (MESD) queue algorithm [JP05]. The MESD algorithm builds on the SESD queues by assigning one local SESD queue to each enqueuer. The local queues are leafs in a shared binary tree, where all threads propagate the minimum timestamp of the front elements towards the root, such that the dequeuer can easily identify the front element in the shared queue element.

  Since the MESD algorithm uses the SESD queue as a building block, we can use the specifications that we have proved in this report in the verification of the MESD algorithm.

  However, we will need additional complicated reasoning about the MESD algorithm; for example the operations on the shared binary tree (parent, children, etc.), correctness of the propagation of the timestamp to the root of the tree, and additional shared variables such as the counter for obtaining timestamps. Furthermore, the MESD algorithm uses a load-linked/store-conditional (LL/SC) operation. Since HeapLang does not support LL/SC operations, we could either implement it using the compare-and-swap (CAS) primitive, or extend HeapLang with LL/SC as an atomic primitive.

References

[BB23] Lars Birkedal and Aleš Bizjak. Lecture notes on iris: Higher-order concurrent separation logic, 2023.

[JKJ+18] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018.

[JP05] Prasad Jayanti and Srdjan Petrovic. Logarithmic-time single deleter, multiple inserter wait-free queues and stacks. In Sundar Sarukkai and Sandeep Sen, editors, *FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science*, pages 408–419, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[MS96] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, page 267–275, New York, NY, USA, 1996. Association for Computing Machinery.

[NBT26] Egor Namakonov, Lars Birkedal, and Amin Timany. Verifying wait-freedom for concurrent higher-order programs. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP 2026)*, 2026. Technical paper.

[Ped24] Mathias Pedersen. Verification of the blocking and non-blocking michael-scott queue algorithms. Master's thesis, Aarhus University, 2024.

[SBP13] Kasper Svendsen, Lars Birkedal, and Matthew Parkinson. Modular reasoning about separation of concurrent data structures. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, pages 169–188, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[The26] The Rocq Development Team. The rocq prover, 2026.

[VB21] Simon Friis Vindum and Lars Birkedal. Contextual refinement of the michael-scott queue (proof pearl). In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2021, page 76–90, New York, NY, USA, 2021. Association for Computing Machinery.

A Concurrent Setting With Freeing: Detailed Definitions

A.1 Freeing the Location in FreeLater

DequeueFreeFreeLaterPtNone

$$\frac{\mathsf{logicalPts}(\gamma_{\mathsf{pt}},\gamma_{\mathsf{dom}},\ell_{\mathsf{First}},\ell_{\mathsf{next}},\mathsf{tbfCandidate}(\ell_{\mathsf{FL}}))\quad \ell_{\mathsf{FL}}\mapsto_{1/2}\mathsf{inj}_1\,()\quad \ell_{\mathsf{FL}}\overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{1/2}\mathsf{inj}_1\,()\quad \ell_{\mathsf{First}}\overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{\square}\mathsf{inj}_2\,((ts,tid,w),\mathsf{next})\quad \mathsf{next}\overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{\square}\ell_{\mathsf{next}}\quad \ulcorner\ell_{\mathsf{First}}\neq\ell_{\mathsf{FL}}\urcorner\quad \ell_{\mathsf{dummy}}\mapsto_{1/2}\mathsf{inj}_1\,()\quad \boxed{\circ\{\ell_{\mathsf{dummy}}\}}^{\gamma_{\mathsf{dom}}}\quad \ell_{\mathsf{next}}\overset{\gamma_{\mathsf{dom}},\gamma_{\mathsf{pt}}}{\rightsquigarrow}\ell_{\mathsf{dummy}}}{\ell_{\mathsf{FL}}\mapsto\mathsf{inj}_1\,()*\left(\mathsf{freed}(\ell_{\mathsf{FL}})\mathrel{-\!\!*}\Rrightarrow\mathsf{logicalPts}(\gamma_{\mathsf{pt}},\gamma_{\mathsf{dom}},\ell_{\mathsf{First}},\ell_{\mathsf{next}},\mathsf{tbfNone})*\ell_{\mathsf{dummy}}\mapsto_{1/2}\mathsf{inj}_1\,()\right)}$$

DequeueFreeFreeLaterPtPers

$$\frac{\mathsf{logicalPts}(\gamma_{\mathsf{pt}},\gamma_{\mathsf{dom}},\ell_{\mathsf{First}},\ell_{\mathsf{next}},\mathsf{tbfCandidate}(\ell_{\mathsf{FL}}))\quad \ell_{\mathsf{FL}}\overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{\square}\mathsf{inj}_2\,((ts',tid',w'),\mathsf{next}_{\mathsf{FL}})\quad \mathsf{ptNoneOrPtPersReach}(\ell_{\mathsf{First}},\ell_{\mathsf{next}},\gamma_{\mathsf{dom}},\gamma_{\mathsf{pt}})\quad \ell_{\mathsf{FL}}\overset{\gamma_{\mathsf{dom}},\gamma_{\mathsf{pt}}}{\rightsquigarrow}\ell_{\mathsf{next}}\quad \ell_{\mathsf{next}}\overset{\gamma_{\mathsf{dom}},\gamma_{\mathsf{pt}}}{\rightsquigarrow}\ell_{\mathsf{dummy}}\quad \ell_{\mathsf{dummy}}\mapsto_{1/2}\mathsf{inj}_1\,()\quad \boxed{\circ\{\ell_{\mathsf{dummy}}\}}^{\gamma_{\mathsf{dom}}}\quad \ulcorner\ell_{\mathsf{FL}}\neq\ell_{\mathsf{next}}\urcorner\quad \ulcorner\ell_{\mathsf{FL}}\neq\ell_{\mathsf{First}}\urcorner\quad \ulcorner\ell_{\mathsf{FL}}\neq\mathsf{next}_{\mathsf{FL}}\urcorner}{\ell_{\mathsf{FL}}\mapsto\mathsf{inj}_1\,()*\left(\mathsf{freed}(\ell_{\mathsf{FL}})\mathrel{-\!\!*}\Rrightarrow\mathsf{logicalPts}(\gamma_{\mathsf{pt}},\gamma_{\mathsf{dom}},\ell_{\mathsf{First}},\ell_{\mathsf{next}},\mathsf{tbfFreeing}(\ell_{\mathsf{FL}}))*\ell_{\mathsf{dummy}}\mapsto_{1/2}\mathsf{inj}_1\,()*\mathsf{ptNoneOrPtPersReach}(\ell_{\mathsf{First}},\ell_{\mathsf{next}},\gamma_{\mathsf{dom}},\gamma_{\mathsf{pt}})\right)}$$

DequeueFreeFreeLaterNext

$$\frac{\mathsf{logicalPts}(\gamma_{\mathsf{pt}},\gamma_{\mathsf{dom}},\ell_{\mathsf{FL}},\ell_{\mathsf{next}},\mathsf{tbfFreeing}(\ell_{\mathsf{First}}))\quad \ell_{\mathsf{FL}}\overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{\square}\mathsf{inj}_2\,((ts',tid',w'),\mathsf{next}_{\mathsf{FL}})\quad \mathsf{next}_{\mathsf{FL}}\overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{\square}\ell'_{\mathsf{next}}\quad \ell_{\mathsf{First}}\overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{\square}\mathsf{inj}_2\,((ts,tid,w),\mathsf{next})\quad \mathsf{next}\overset{\gamma_{\mathsf{pt}}}{\longmapsto}_{\square}\ell_{\mathsf{next}}\quad \ell_{\mathsf{First}}\overset{\gamma_{\mathsf{dom}},\gamma_{\mathsf{pt}}}{\rightsquigarrow}\ell_{\mathsf{next}}\quad \ell_{\mathsf{FL}}\overset{\gamma_{\mathsf{dom}},\gamma_{\mathsf{pt}}}{\rightsquigarrow}\ell_{\mathsf{First}}\quad \ell_{\mathsf{First}}\overset{\gamma_{\mathsf{dom}},\gamma_{\mathsf{pt}}}{\rightsquigarrow}\ell_{\mathsf{dummy}}\quad \ell_{\mathsf{dummy}}\mapsto_{1/2}\mathsf{inj}_1\,()\quad \boxed{\circ\{\ell_{\mathsf{dummy}}\}}^{\gamma_{\mathsf{dom}}}\quad \ulcorner\ell_{\mathsf{First}}\neq\ell_{\mathsf{FL}}\urcorner\quad \ulcorner\ell_{\mathsf{FL}}\neq\ell_{\mathsf{next}}\urcorner}{(\exists v,\ \mathsf{next}_{\mathsf{FL}}\mapsto v)\ *\ \left(\mathsf{freed}(\mathsf{next}_{\mathsf{FL}})\mathrel{-\!\!*}\Rrightarrow\mathsf{logicalPts}(\gamma_{\mathsf{pt}},\gamma_{\mathsf{dom}},\ell_{\mathsf{FL}},\ell_{\mathsf{next}},\mathsf{tbfNone})*\ell_{\mathsf{dummy}}\mapsto_{1/2}\mathsf{inj}_1\,()\right)}$$

A.2 The Announced Location in the Invariant

We have added the following propositions to the invariant to handle the communication about the announced location:

$$\begin{aligned}&\mathsf{queueInv}(\mathsf{First},\mathsf{Last},\mathsf{FreeLater},\mathsf{Announce},\mathsf{Help},G,\Phi)\triangleq\\&\quad\exists\ell_{\mathsf{First}},\ell_{\mathsf{Last}},\ell^?_{\mathsf{A}},\ell_{\mathsf{FL}},v^?_{\mathsf{Help}},\ell_{\mathsf{dummy}},\ell_{\mathsf{init}},xs,\mathsf{tbf},\ \ell^?_{\mathsf{fstAtAnno}},\ell^?_{\mathsf{X\text{-}A}},\ell^?_{\mathsf{X\text{-}fstAtAnno}},\ell_{\mathsf{not\text{-}fstAtAnno}},\mathsf{S}_{pastAnnos}\ .\\&\quad\ldots\\&\quad\boxed{\tfrac{1}{2}(\ell^?_{\mathsf{X\text{-}A}},\ell^?_{\mathsf{X\text{-}fstAtAnno}})}^{\gamma_{\mathsf{X}}}*\boxed{\tfrac{1}{2}(\ell^?_{\mathsf{fstAtAnno}})}^{\gamma_{\mathsf{FstAtAnno}}}*\\&\quad\boxed{\tfrac{1}{2}(\ell_{\mathsf{not\text{-}fstAtAnno}})}^{\gamma_{\mathsf{not\text{-}fstAtAnno}}}*\ulcorner\ell_{\mathsf{First}}\neq\ell_{\mathsf{not\text{-}fstAtAnno}}\urcorner*\\&\quad\mathsf{fstAtAnnoDisj}(\ell^?_{\mathsf{fstAtAnno}},\ell^?_{\mathsf{A}},\ell^?_{\mathsf{X\text{-}A}},\ell^?_{\mathsf{X\text{-}fstAtAnno}},\ell_{\mathsf{First}},\ell_{\mathsf{not\text{-}fstAtAnno}})*\\&\quad\mathsf{fstAtAnnoAnnoImpl}(\ell^?_{\mathsf{fstAtAnno}},\ell^?_{\mathsf{A}},\ell_{\mathsf{First}},\mathsf{tbf},\ell_{\mathsf{FL}},\gamma_{\mathsf{dom}},\gamma_{\mathsf{pt}})*\\&\quad\boxed{\bullet\mathsf{S}_{pastAnnos}}^{\gamma_{\mathsf{pastAnnos}}}*\mathsf{pastAnnosDisj}(\mathsf{S}_{pastAnnos},\ell^?_{\mathsf{A}},\ell_{\mathsf{First}},\gamma_{\mathsf{pastAnnos}},\gamma_{\mathsf{dom}},\gamma_{\mathsf{pt}})\ .\end{aligned}$$

We give the precise definitions.

$\boxed{\frac{1}{2}(\ell^?_{\mathsf{X\text{-}A}},\ell^?_{\mathsf{X\text{-}fstAtAnno}})}^{\gamma_{\mathsf{X}}}$, $\boxed{\frac{1}{2}(\ell^?_{\mathsf{fstAtAnno}})}^{\gamma_{\mathsf{FstAtAnno}}}$ and $\boxed{\frac{1}{2}(\ell_{\mathsf{not\text{-}fstAtAnno}})}^{\gamma_{\mathsf{not\text{-}fstAtAnno}}}$ are all defined using the fractional agreement RA over pairs of option locations, option locations and locations respectively. This RA satisfies agreement of the value, and we can update the value by owning the entire fraction, 1.
The enqueuer owns the other half of $\boxed{\frac{1}{2}(\ell^?_{\mathsf{fstAtAnno}})}^{\gamma_{\mathsf{FstAtAnno}}}$, while the dequeuer owns the other halves of $\boxed{\frac{1}{2}(\ell^?_{\mathsf{X\text{-}A}},\ell^?_{\mathsf{X\text{-}fstAtAnno}})}^{\gamma_{\mathsf{X}}}$ and $\boxed{\frac{1}{2}(\ell_{\mathsf{not\text{-}fstAtAnno}})}^{\gamma_{\mathsf{not\text{-}fstAtAnno}}}$. This division is decided based on which of the two threads will update the values.

We define the following disjunction for $\ell^?_{\mathsf{fstAtAnno}}$:

$$\begin{aligned}&\mathsf{fstAtAnnoDisj}(\ell^?_{\mathsf{fstAtAnno}}, \ell^?_{\mathsf{A}}, \ell^?_{\mathsf{X\text{-}A}}, \ell^?_{\mathsf{X\text{-}fstAtAnno}}, \ell_{\mathsf{First}}, \ell_{\mathsf{not\text{-}fstAtAnno}}) \triangleq \\ &\quad \big(\ulcorner \ell^?_{\mathsf{fstAtAnno}} = \texttt{None} \urcorner \vee (\exists \ell_{\mathsf{fstAtAnno}}, \ulcorner \ell^?_{\mathsf{fstAtAnno}} = \texttt{Some}(\ell_{\mathsf{fstAtAnno}}) \urcorner * \ulcorner \ell_{\mathsf{fstAtAnno}} \neq \ell_{\mathsf{not\text{-}fstAtAnno}} \urcorner)\big) \\ &\quad \vee\ (\ulcorner \ell^?_{\mathsf{A}} = \ell^?_{\mathsf{X\text{-}A}} \urcorner * \ulcorner \ell^?_{\mathsf{fstAtAnno}} = \ell^?_{\mathsf{X\text{-}fstAtAnno}} \urcorner).\end{aligned}$$

Either we are in the exception case, where we don't know anything about $\ell^?_{\mathsf{fstAtAnno}}$, or it is `None`, or it is some location $\ell_{\mathsf{fstAtAnno}}$ that is *not* equal to $\ell_{\mathsf{not\text{-}fstAtAnno}}$.