Master's Thesis Exam
Verification of the Blocking and Non-Blocking Michael-Scott Queue Algorithms

Mathias Pedersen, 201808137
Advisor: Amin Timany

Aarhus University

June 2024

AARHUS UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Overview of the Project and Contributions

- Initial goal was to prove soundness of the two M&S Queues
- The project later generalised the results to apply to queues in general
- In particular, three different specifications for queues were given
  - Sequential specification
    - Useful for sequential clients
  - Concurrent specification
    - Proves soundness of concurrent queues
    - Useful for some concurrent clients
  - HOCAP-style specification
    - Stronger specification, useful for more complex clients
    - Demonstrated with a specific queue client (queueAdd)
- It was demonstrated that the HOCAP-style specification derives the other two specifications
- Implementations of the M&S Queues in HeapLang were proven to meet the three specifications
  - In particular, both version are sound
- All proofs have been mechanised in the Coq proof assistant

Outline

Queue Specifications

Specifications for Queues

Assumptions on Queues

- Queues consists of initialize, enqueue, and dequeue
- initialize creates an empty queue: $[]$
- enqueue adds a value, v, to the beginning of the queue xs_v: $v :: xs_v$
- dequeue depends on whether queue is empty:
  - If non-empty, $xs_v ++ v$, remove v and return Some v
  - If empty, $[]$, return None

Nature of Specifications

- Specifications written in Iris, a higher order CSL
- Expressed in terms of *Hoare triples*: $\{P\}\, e\, \{v.\Phi\ v\}$
- Hoare triples prove partial correctness of programs, e
- In particular: safety
- Idea: clients can use Hoare triples to prove results about their own code

Sequential Specification

Definition (Sequential Specification)

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

- The proposition $\mathrm{isQueue}_S(v_q, xs_v, G)$, states that value v_q represents the queue, which contains elements xs_v
- $G \in \mathit{SeqQgnames}$ is a collection of ghost names (depends on specific queue)
- Specification consists of three Hoare triples – one for each queue function
- Important: $\mathrm{isQueue}_S$ not required to be persistent!

Concurrent Specification

- To support concurrent clients, we shall require the queue predicate be persistent
- Tracking the contents of queue in the way that the sequential specification did doesn't work
- Threads will start disagreeing on contents of queue, as they have only local view of contents
- Give up on tracking contents for now
- Instead, promise that all elements satisfy client-defined predicate, Ψ

Definition (Concurrent Specification)

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

HOCAP-style Specification - Abstract State RA

- We will need a construction to allow clients to track contents of queue
- Idea: have two "views" of the abstract state of the queue

Authoritative view	**Fragmental view**
$\gamma \Mapsto_\bullet xs_v$	$\gamma \Mapsto_\circ xs_v$
Owned by queue	Owned by client

- Construction ensures:
  - authoritative and fragmental views always agree on abstract state of queue
  - views can only be updated in unison
- Implemented using the resource algebra: $\textsc{Auth}((\textsc{Frac} \times \textsc{Ag}(\mathit{List\ Val}))^?)$
- The desirables are captured by the following lemmas

Lemmas on the Abstract State RA

$$\vdash \Rrightarrow \exists \gamma.\, \gamma \Mapsto_\bullet xs_v * \gamma \Mapsto_\circ xs_v \qquad \text{(Abstract State Alloc)}$$

$$\gamma \Mapsto_\bullet xs_v' * \gamma \Mapsto_\circ xs_v \vdash xs_v = xs_v' \qquad \text{(Abstract State Agree)}$$

$$\gamma \Mapsto_\bullet xs_v' * \gamma \Mapsto_\circ xs_v \Rrightarrow \gamma \Mapsto_\bullet xs_v'' * \gamma \Mapsto_\circ xs_v'' \qquad \text{(Abstract State Update)}$$

HOCAP-style Specification

- Post-condition of initialize specification now gives fragmental view to clients
- Hoare triples for enqueue and dequeue are conditioned on view-shifts
- Clients must show that they can supply the fragmental view, so that the abstract (and concrete) state can be updated
- View-shifts and Hoare-triples parametrised by predicates P and Q
  - Client might have resources that need to be updated as a result of enqueue/dequeue
  - P is the clients resources before enqueue/dequeue and Q the resources after

Definition (HOCAP Specification)

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

Queue Client - A PoC Client

- Idea: a minimal client complex enough to require HOCAP specification
- Uses parallel composition, so sequential specification insufficient
- Relies on dequeues not returning None, so concurrent specification insufficient
- HOCAP specification supports consistency and allows us to track queue contents, allowing us to exclude cases where dequeue returns None

$$\text{unwrap}\ \ w \triangleq \mathbf{match}\ w\ \mathbf{with}\ \text{None} \Rightarrow ()\ ()\ |\ \text{Some}\ v \Rightarrow v\ \mathbf{end}$$

$$\text{enqdeq}\ \ v_q\ \ c \triangleq \text{enqueue}\ \ v_q\ \ c; \text{unwrap}(\text{dequeue}\ \ v_q)$$

$$\begin{aligned}
&\text{queueAdd}\ \ a\ b \triangleq \\
&\quad \mathbf{let}\ v_q = \text{initialize}\ ()\ \mathbf{in} \\
&\quad \mathbf{let}\ p = (\text{enqdeq}\ \ v_q\ \ a) \,||\, (\text{enqdeq}\ \ v_q\ \ b)\ \mathbf{in} \\
&\quad \mathbf{fst}\ p + \mathbf{snd}\ p
\end{aligned}$$

Queue Client - A PoC Client (continued)

Lemma (QueueAdd Specification)

$$\forall a, b \in \mathbb{Z}. \{True\} \text{ queueAdd } a \; b \, \{v. v = a + b\}$$

- Proof idea: Create invariant capturing possible states of queue contents
- Tokens are used to reason about which state we are in

Definition (Invariant for QueueAdd)

$$\begin{aligned}
I_{QA}(G, Ga, a, b) \triangleq\ & G.\gamma_{\mathrm{Abst}} \Mapsto_{\circ} [] \ * \ \mathrm{TokD1}\ \ Ga \ * \ \mathrm{TokD2}\ \ Ga \vee \\
& G.\gamma_{\mathrm{Abst}} \Mapsto_{\circ} [a] \ * \ \mathrm{TokA}\ \ Ga \ * \ (\mathrm{TokD1}\ \ Ga \ \vee \ \mathrm{TokD2}\ \ Ga) \vee \\
& G.\gamma_{\mathrm{Abst}} \Mapsto_{\circ} [b] \ * \ \mathrm{TokB}\ \ Ga \ * \ (\mathrm{TokD1}\ \ Ga \ \vee \ \mathrm{TokD2}\ \ Ga) \vee \\
& G.\gamma_{\mathrm{Abst}} \Mapsto_{\circ} [a; b] \ * \ \mathrm{TokA}\ \ Ga \ * \ \mathrm{TokB}\ \ Ga \vee \\
& G.\gamma_{\mathrm{Abst}} \Mapsto_{\circ} [b; a] \ * \ \mathrm{TokB}\ \ Ga \ * \ \mathrm{TokA}\ \ Ga \vee
\end{aligned}$$

The Two-Lock Michael-Scott Queue

Implementation: initialize

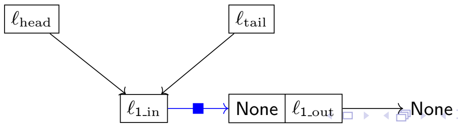
▶format◀

- The data structure is a linked list
- A node x in the linked list is a triple, $x = (\ell_{\text{in}}, w, \ell_{\text{out}})$, with ℓ_{in} pointing to (w, ℓ_{out})
- We use the following notation for nodes

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

- The initialize function first creates an initial head node, x_{head}
- Then a lock protecting the head pointer, and a lock protecting the tail pointer
- Finally, it creates the head and tail pointers, ℓ_{head} and ℓ_{tail}, both pointing to x_{head}

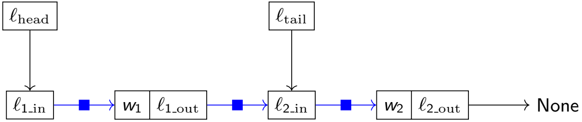
$$
\begin{array}{l}
\text{initialize} \triangleq \\
\quad \textbf{let } \mathit{node} = \textbf{ref}\,(\text{None}, \textbf{ref}\,(\text{None}))\ \textbf{in} \\
\quad \textbf{let } \mathit{H\_lock} = \text{newLock}()\ \textbf{in} \\
\quad \textbf{let } \mathit{T\_lock} = \text{newLock}()\ \textbf{in} \\
\quad \textbf{ref}\,((\textbf{ref}\,(\mathit{node}), \textbf{ref}\,(\mathit{node})), (\mathit{H\_lock}, \mathit{T\_lock}))
\end{array}
$$



Implementation: enqueue

- The enqueue function consists of the following steps
  1. Create a new node, x_{new}, containing value to be enqueued
  2. Acquire the tail lock
  3. Add x_{new} to linked list
  4. Swing tail pointer to x_{new}
  5. Release the tail lock

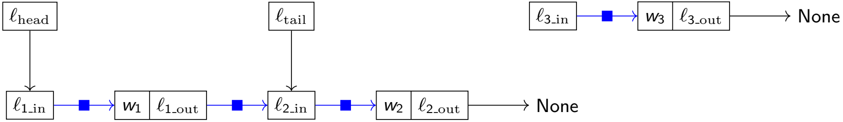
$$
\begin{aligned}
&\mathrm{enqueue}\ \ Q\ \mathit{value} \triangleq \\
&\quad \mathbf{let}\ \mathit{node} = \mathsf{ref}\,(\mathrm{Some}\ \mathit{value}, \mathsf{ref}\,(\mathrm{None}))\ \mathbf{in} \\
&\quad \mathrm{acquire}(\mathsf{snd}(\mathsf{snd}(!\,Q))); \\
&\quad \mathsf{snd}(!(!(\mathsf{snd}(\mathsf{fst}(!\,Q)))))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



Implementation: enqueue

- The enqueue function consists of the following steps
  1. Create a new node, x_{new}, containing value to be enqueued
  2. Acquire the tail lock
  3. Add x_{new} to linked list
  4. Swing tail pointer to x_{new}
  5. Release the tail lock

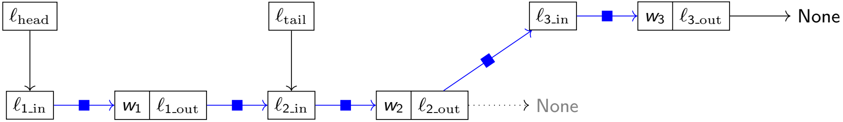
$$
\begin{array}{l}
\text{enqueue}\ \ Q\ \mathit{value} \triangleq \\
\quad \mathsf{let}\ \mathit{node} = \mathsf{ref}\,(\text{Some}\ \mathit{value}, \mathsf{ref}\,(\text{None}))\ \mathsf{in} \\
\quad \text{acquire}(\mathsf{snd}(\mathsf{snd}(!\,Q))); \\
\quad \mathsf{snd}(!(!(\mathsf{snd}(\mathsf{fst}(!\,Q))))) \leftarrow \mathit{node}; \\
\quad \mathsf{snd}(\mathsf{fst}(!\,Q)) \leftarrow \mathit{node}; \\
\quad \text{release}(\mathsf{snd}(\mathsf{snd}(!\,Q)))
\end{array}
$$



Implementation: enqueue

- The enqueue function consists of the following steps
  1. Create a new node, x_{new}, containing value to be enqueued
  2. Acquire the tail lock
  3. Add x_{new} to linked list
  4. Swing tail pointer to x_{new}
  5. Release the tail lock

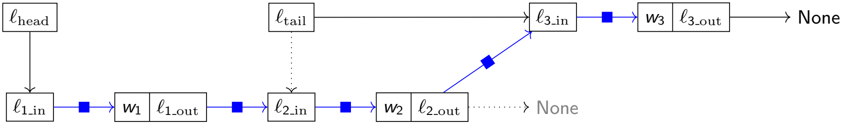
$$
\begin{array}{l}
\mathsf{enqueue}\ \ Q\ \mathit{value} \triangleq \\
\quad \mathsf{let}\ \mathit{node} = \mathsf{ref}\,(\mathsf{Some}\ \mathit{value}, \mathsf{ref}\,(\mathsf{None}))\ \mathsf{in} \\
\quad \mathsf{acquire}(\mathsf{snd}(\mathsf{snd}(!\,Q))); \\
\quad \mathsf{snd}(!(!(\mathsf{snd}(\mathsf{fst}(!\,Q))))) \leftarrow \mathit{node}; \\
\quad \mathsf{snd}(\mathsf{fst}(!\,Q)) \leftarrow \mathit{node}; \\
\quad \mathsf{release}(\mathsf{snd}(\mathsf{snd}(!\,Q)))
\end{array}
$$



Implementation: enqueue

- The enqueue function consists of the following steps
  1. Create a new node, x_{new}, containing value to be enqueued
  2. Acquire the tail lock
  3. Add x_{new} to linked list
  4. Swing tail pointer to x_{new}
  5. Release the tail lock

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

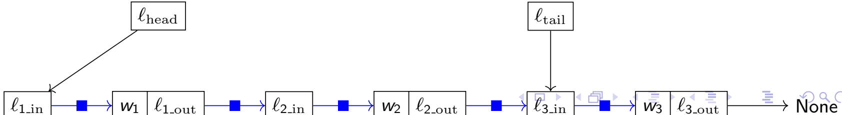


Implementation: dequeue

▶format◀

- The dequeue function checks if queue is empty
  - If empty, return *None*
  - Else, swing head pointer to new head, and return dequeued value

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

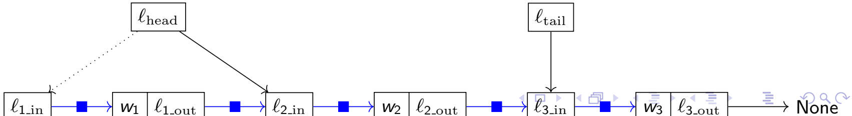


Implementation: dequeue

▶format◀

- The dequeue function checks if queue is empty
  - If empty, return *None*
  - Else, swing head pointer to new head, and return dequeued value

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



Observations on Behaviour of the Two-Lock M&S Queue

▶format and simplify◀

1. The tail node is always either the last or second last node in the linked list.
2. All but the last pointer in the linked list (the pointer to None) never change.
3. Nodes in the linked list are never deleted. Hence, the linked list only ever grows.
4. The tail can lag one node behind the head.
5. At any given time, the queue is in one of four states:
    1. No threads are interacting with the queue (**Static**).
    2. A thread is enqueueing (**Enqueue**).
    3. A thread is dequeueing (**Dequeue**).
    4. A thread is enqueueing and a thread is dequeueing (**Both**).

Proving that the Two-Lock Michael-Scott Queue Satisfies the HOCAP-style Specification

The isLL Predicate

format slide

- Idea: express the structure of the linked list in terms of points-to predicates
- Also captures persistent and non-persistent parts of the linked list

Definition (Linked List Chain Predicate)

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

Definition (Linked List Predicate)

$$\begin{aligned}
\mathsf{isLL}([]) &\triangleq \mathsf{True} \\
\mathsf{isLL}(x :: xs) &\triangleq \mathsf{out}(x) \mapsto \mathsf{None} * \mathsf{isLL_chain}(x :: xs)
\end{aligned}$$

Example

Invariant

▶format slide◀

- Queue predicate must be persistent (according to specification)
- The queue relies on non-persistent resources (e.g. $\ell_{\mathrm{head}} \mapsto \ell_{\mathrm{in}}$)
- Solution: identify a *queue invariant*, describing the resources
- Invariants are persistent in Iris

Definition (Two-Lock M&S Queue HOCAP Invariant)

$$\begin{aligned}
&\mathsf{I}_{\mathsf{TLH}}(\ell_{\mathrm{head}}, \ell_{\mathrm{tail}}, G) \triangleq \\
&\exists xs_v.\ G.\gamma_{\mathrm{Abst}} \Rightarrow_{\bullet} xs_v * \\
&\exists xs, xs_{\mathrm{queue}}, xs_{\mathrm{old}}, x_{\mathrm{head}}, x_{\mathrm{tail}}. \\
&xs = xs_{\mathrm{queue}} +\!\!+ [x_{\mathrm{head}}] +\!\!+ xs_{\mathrm{old}} * \\
&\mathsf{isLL}(xs) * \\
&\mathsf{projVal}(xs_{\mathrm{queue}}) = \mathsf{wrapSome}(xs_v) * \\
&(\\
&\quad \ell_{\mathrm{head}} \mapsto \mathsf{in}(x_{\mathrm{head}}) * \ell_{\mathrm{tail}} \mapsto \mathsf{in}(x_{\mathrm{tail}}) * \mathsf{isLast}(x_{\mathrm{tail}}, xs) * && \textbf{(Static)} \\
&\quad \mathsf{TokNE}\ \ G * \mathsf{TokND}\ \ G * \mathsf{TokUpdated}\ \ G \\
&\ \vee
\end{aligned}$$

Queue Predicate

- HOCAP-style specification requires the existence of a persistent queue predicate
- We define it in terms of our invariant

Definition (Two-Lock M&S Queue - isQueue Predicate)

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

- The queue predicate is persistent, as all its constituents are
- Proving that TLMSQ satisfies the HOCAP-style specification then consists of proving the Hoare triples for initialize, enqueue, and dequeue
- We here focus on enqueue

Proof Sketch of the Hoare triple for enqueue

▶**format**◀ Must prove:

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

Assume the view-shift, and the persistent information in $\mathsf{isQueue}(v_q, Qgnames)$: $v_q = \ell_{\mathrm{queue}} * \ell_{\mathrm{queue}} \mapsto^{\square} ((\ell_{\mathrm{head}}, \ell_{\mathrm{tail}}), (h_{\mathrm{lock}}, t_{\mathrm{lock}}))$, the invariant $\boxed{\mathsf{I}_{\mathsf{TLH}}(\ell_{\mathrm{head}}, \ell_{\mathrm{tail}}, G)}^{\mathcal{N}.queue}$, and $\mathsf{isLock}(G.\gamma_{\mathrm{Tlock}}, t_{\mathrm{lock}}, \mathsf{TokE}\ \ G)$

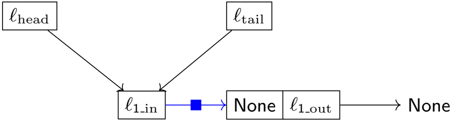
$\{P\}$
  let $node$ = ref (Some v, ref (None)) in   (create node x_{new})
$\{P * \mathsf{out}(x_{\mathrm{new}}) \mapsto \mathsf{None}\}$
  acquire(snd(snd(! v_q)));   (acquire tail lock)
$\{P * \mathsf{out}(x_{\mathrm{new}}) \mapsto \mathsf{None} * \mathsf{TokE}\ \ G\}$
  e_t = !(snd(fst(! v_q)))   (find current tail, x_{tail}. $\mathsf{I}_{\mathsf{TLH}}$: **Static**/**Dequeue** → **Enqueue**/**Both** (before))
$\{P * \mathsf{out}(x_{\mathrm{new}}) \mapsto \mathsf{None} * \ell_{\mathrm{tail}} \mapsto^{\frac{1}{2}} \mathsf{in}(x_{\mathrm{tail}}) * \mathsf{TokNE}\ \ G * \mathsf{TokAfter}\ \ G\}$
  snd(!(e_t)) ← $node$;   (make x_{tail} point to x_{new}. $\mathsf{I}_{\mathsf{TLH}}$: **Enqueue**/**Both** (before) → **Enqueue**/**Both** (after))
$\{Q * \ell_{\mathrm{tail}} \mapsto^{\frac{1}{2}} \mathsf{in}(x_{\mathrm{tail}}) * \mathsf{TokNE}\ \ G * \mathsf{TokBefore}\ \ G\}$
  snd(fst(! v_q)) ← $node$;   (swing tail pointer to x_{new}. $\mathsf{I}_{\mathsf{TLH}}$: **Enqueue**/**Both** (after) → **Static**/**Dequeue**)
$\{Q * \mathsf{TokE}\ \ G\}$
  release(snd(snd(! v_q)))   (release tail lock)
$\{Q\}$

The Lock-Free Michael-Scott Queue

Implementation: initialize

- Queue data structure is still a linked list
- The lock-free versions of initialize, enqueue, and dequeue perform the same manipulations of the linked list as two-lock versions
- Difference is how the manipulations take place: CAS
- No longer need locks

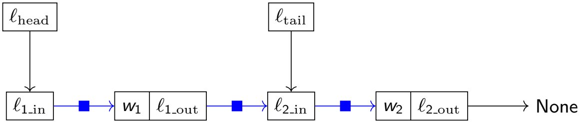
$$\begin{array}{l} \mathsf{initialize} \triangleq \\ \quad \mathsf{let}\ \mathit{node} = \mathsf{ref}\,(\mathsf{None}, \mathsf{ref}\,(\mathsf{None}))\ \mathsf{in} \\ \quad \mathsf{ref}\,(\mathsf{ref}\,(\mathit{node}), \mathsf{ref}\,(\mathit{node})) \end{array}$$



Implementation: enqueue

- Appending x_{new} to linked list is now done with CAS
- Ensures that no other thread has performed an enqueue while we have been working
- Swinging tail to x_{new} might fail: another thread has helped us

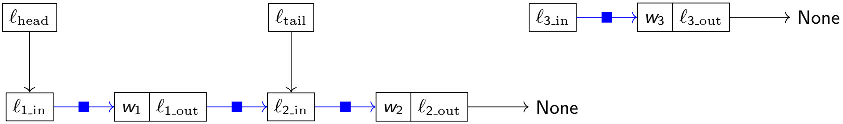
```
enqueue  Q value ≜
  let node = ref (Some value, ref (None)) in
  (rec  loop_ =
    let tail = !(snd(! Q)) in
    let next = !(snd(! tail)) in
    if tail = !(snd(! Q)) then
      if next = None then
        if CAS (snd(! tail)) next node then
          CAS (snd(! Q)) tail node
        else loop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



Implementation: enqueue

- Appending x_{new} to linked list is now done with CAS
- Ensures that no other thread has performed an enqueue while we have been working
- Swinging tail to x_{new} might fail: another thread has helped us

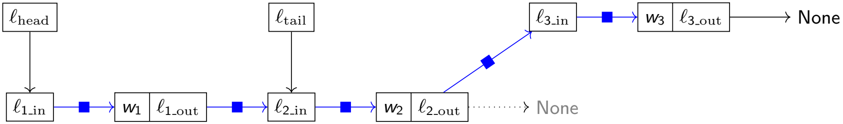
```
enqueue  Q value ≜
  let node = ref (Some value, ref (None)) in
  (rec  loop_ =
    let tail = !(snd(! Q)) in
    let next = !(snd(! tail)) in
    if tail = !(snd(! Q)) then
      if next = None then
        if CAS (snd(! tail)) next node then
          CAS (snd(! Q)) tail node
        else loop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



Implementation: enqueue

- Appending x_{new} to linked list is now done with CAS
- Ensures that no other thread has performed an enqueue while we have been working
- Swinging tail to x_{new} might fail: another thread has helped us

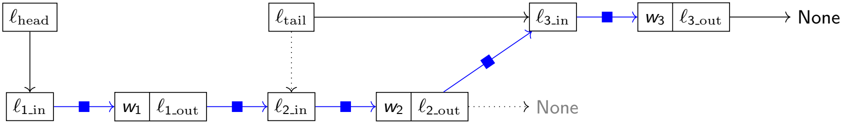
```
enqueue  Q value ≜
  let node = ref (Some value, ref (None)) in
  (rec  loop_ =
    let tail = !(snd(! Q)) in
    let next = !(snd(! tail)) in
    if tail = !(snd(! Q)) then
      if next = None then
        if CAS (snd(! tail)) next node then
          CAS (snd(! Q)) tail node
        else loop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



Implementation: enqueue

- Appending x_{new} to linked list is now done with CAS
- Ensures that no other thread has performed an enqueue while we have been working
- Swinging tail to x_{new} might fail: another thread has helped us

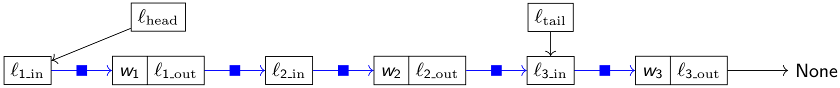
```
enqueue  Q value ≜
  let node = ref (Some value, ref (None)) in
  (rec  loop_ =
    let tail = !(snd(! Q)) in
    let next = !(snd(! tail)) in
    if tail = !(snd(! Q)) then
      if next = None then
        if CAS (snd(! tail)) next node then
          CAS (snd(! Q)) tail node
        else loop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



Implementation: dequeue

- Head now swung with CAS
- Ensures that another thread hasn't dequeued the element we are trying to dequeue

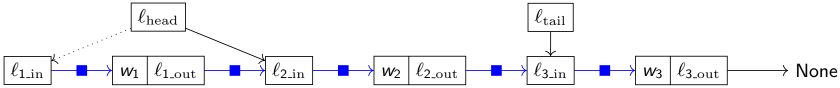
```
dequeue Q ≜
  (rec loop_ =
    let head = !(fst(! Q)) in
    let tail = !(snd(! Q)) in
    let p = newproph in
    let next = !(snd(! head)) in
    if head = Resolve(!(fst(! Q)), p, ()) then
      if head = tail then
        if next = None then
          None
        else
          CAS(snd(! Q))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



Implementation: dequeue

- Head now swung with CAS
- Ensures that another thread hasn't dequeued the element we are trying to dequeue

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



Prophecies

►create slide◄

The Lock-and-CC-Free Michael-Scott Queue

- Consistency checks and associated loops gone
- Can also remove prophecy in dequeue
  - When we read *next*, we know immediately whether dequeue will conclude empty queue
  - both *head* and *tail* are already fixed

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

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

```
dequeue Q ≜
  (rec loop_ =
    let head = !(fst(! Q)) in
    let tail = !(snd(! Q)) in
    let next = !(snd(! head)) in
    if head = tail then
      if next = None then
        None
      else
        CAS(snd(! Q)) tail next; loop ()
    else
      let value = fst(! next) in
      if CAS (fst(! Q)) head next then
        value
      else loop ()
  )()
```

Proving that the Lock-and-CC-Free Michael-Scott Queue Satisfies the HOCAP-style Specification

Reachability

- The queue relies on some important properties to function correctly:
  - The set of nodes reachable from a particular node only grows
  - The head and tail are only moved forward in the linked list
  - The tail cannot lag behind the head (unlike in the two-lock version)
- We capture all these properties with a notion of *reachability*
- Consists of a concrete and abstract version of reachability

Concrete Reachability

- Concrete reachability essentially captures a section of the linked list (á la isLL)
- The proposition $x_n \rightsquigarrow x_m$ asserts that x_n can reach x_m through the linked list
- Defined inductively as follows

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

- Concrete reachability is reflexive and transitive

Reachability (continued)

Abstract Reachability

- Abstract reachability is concerned with tracking specific *types* of nodes, such as the head node, the tail node, and the last node
- Tracked using ghost names, e.g. γ_{Head}, γ_{Tail}, and γ_{Last}
  - Implemented using the resource algebra $\textsc{Auth}(\mathcal{P}(\mathit{Node}))$
- Defined in two parts: Abstract Points-to ($\gamma \rightarrowtail x$) and Abstract Reach ($x \dashrightarrow \gamma$)
- For instance, $\gamma_{\mathrm{Tail}} \rightarrowtail x_n$ means that the current tail node is x_n
- And $x_m \dashrightarrow \gamma_{\mathrm{Tail}}$ means that node x_m can always reach the tail node

Lemmas for Reachability (simplified)

$$x \rightsquigarrow x \Rrightarrow \exists \gamma.\, \gamma \rightarrowtail x \qquad \text{(Abs Reach Alloc)}$$

$$x_n \dashrightarrow \gamma_m * \gamma_m \rightarrowtail x_m \twoheadrightarrow\!\!* \; x_n \rightsquigarrow x_m \qquad \text{(Abs Reach Concr)}$$

$$x_n \rightsquigarrow x_m * \gamma_m \rightarrowtail x_m \Rrightarrow x_n \dashrightarrow \gamma_m \qquad \text{(Abs Reach Abs)}$$

$$\gamma_m \rightarrowtail x_m * x_m \rightsquigarrow x_o \Rrightarrow \gamma_m \rightarrowtail x_o \qquad \text{(Abs Reach Advance)}$$

In Coq!