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

►**create slide**◄

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)

$$\exists\, \mathsf{isQueue_S} : \mathit{Val} \rightarrow \mathit{List\ Val} \rightarrow \mathit{SeqQgnames} \rightarrow \mathsf{Prop}.$$

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

- The proposition $\mathsf{isQueue_S}(v_q, xs_v, G)$, 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: $\mathsf{isQueue_S}$ not required to be persistent!

Concurrent Specification

Definition (Concurrent Specification)

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

HOCAP-style Specification - Abstract State RA

- ▶**Introduce Auth and Frag predicates for tracking abstract state**◀
- ▶**Show Resource Algebra**◀

Lemmas on the Abstract State RA

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

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

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

HOCAP-style Specification

Definition (HOCAP Specification)

$$\exists\, \mathrm{isQueue} : \mathit{Val} \to \mathit{Qgnames} \to \mathrm{Prop}.$$

$$\forall v_q, G.\ \mathrm{isQueue}(v_q, G) \implies \Box\, \mathrm{isQueue}(v_q, G)$$

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

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

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

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

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

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

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

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)

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

The Two-Lock Michael-Scott Queue

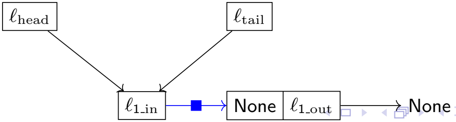
▶format◀

- 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

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

- 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}

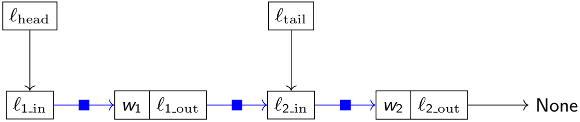
$$
\begin{array}{l}
\mathsf{initialize} \triangleq \\
\quad \mathsf{let}\ node = \mathsf{ref}\,(\mathsf{None}, \mathsf{ref}\,(\mathsf{None}))\ \mathsf{in} \\
\quad \mathsf{let}\ H\_lock = \mathsf{newLock}()\ \mathsf{in} \\
\quad \mathsf{let}\ T\_lock = \mathsf{newLock}()\ \mathsf{in} \\
\quad \mathsf{ref}\,((\mathsf{ref}\,(node),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



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

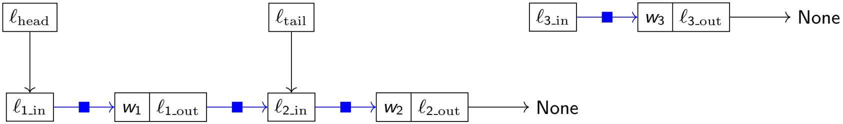
$$
\begin{array}{l}
\mathsf{enqueue}\ \ Q\ \mathit{value} \triangleq \\
\quad \mathsf{let}\ \mathit{node} = \mathsf{ref}\,(\mathsf{Some}\ \mathit{value}, \mathsf{ref}\,(\mathsf{None}))\ \mathsf{in} \\
\quad \mathsf{acquire}(\mathsf{snd}(\mathsf{snd}(!\,Q))); \\
\quad \mathsf{snd}(!(!(\mathsf{snd}(\mathsf{fst}(!\,Q)))))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



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

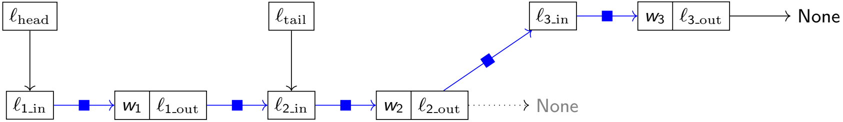
$$
\begin{array}{l}
\text{enqueue}\ \ Q\ \mathit{value} \triangleq \\
\quad \textbf{let}\ \mathit{node} = \textbf{ref}\,(\text{Some}\ \mathit{value}, \textbf{ref}\,(\text{None}))\ \textbf{in} \\
\quad \text{acquire}(\text{snd}(\text{snd}(!\,Q))); \\
\quad \text{snd}(!(!(\text{snd}(\text{fst}(!\,Q)))))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



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

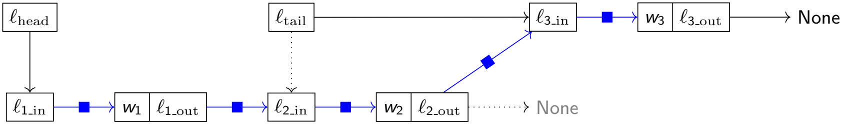
$$
\begin{aligned}
&\mathsf{enqueue}\ \ Q\ \mathit{value} \triangleq \\
&\quad \mathsf{let}\ \mathit{node} = \mathsf{ref}\,(\mathsf{Some}\ \mathit{value}, \mathsf{ref}\,(\mathsf{None}))\ \mathsf{in} \\
&\quad \mathsf{acquire}(\mathsf{snd}(\mathsf{snd}(!\,Q))); \\
&\quad \mathsf{snd}(!(!(\mathsf{snd}(\mathsf{fst}(!\,Q)))))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



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

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

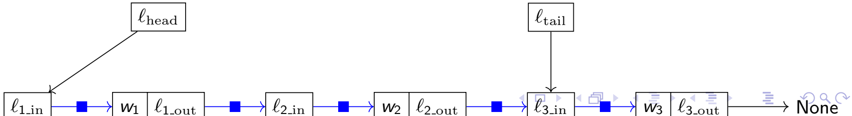


Implementation: dequeue

▶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
```

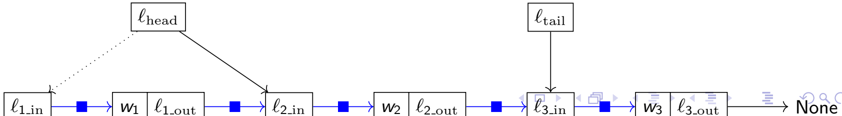


Implementation: dequeue

▶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)

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

Definition (Linked List Predicate)

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

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)

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

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)

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

- 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:

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

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

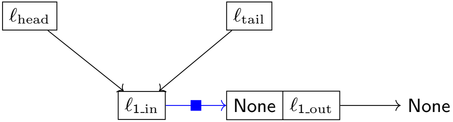
$\{P\}$
  let *node* = ref (Some v, ref (None)) in (create node x_{new})
$\{P * \mathrm{out}(x_{\mathrm{new}}) \mapsto \mathrm{None}\}$
  acquire(snd(snd(! v_q))); (acquire tail lock)
$\{P * \mathrm{out}(x_{\mathrm{new}}) \mapsto \mathrm{None} * \mathrm{TokE}\ \ G\}$
  e_t = !(snd(fst(! v_q))) (find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

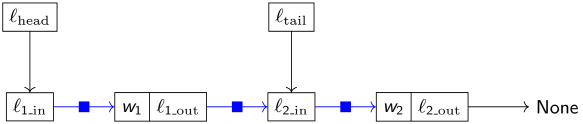
$$\begin{array}{l} \text{initialize} \triangleq \\ \quad \mathsf{let}\ \mathit{node} = \mathsf{ref}\,(\mathsf{None}, \mathsf{ref}\,(\mathsf{None}))\ \mathsf{in} \\ \quad \mathsf{ref}\,(\mathsf{ref}\,(\mathit{node}), \mathsf{ref}\,(\mathit{node})) \end{array}$$



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

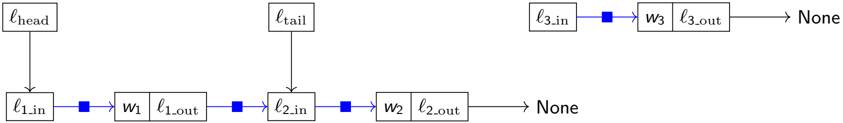
```
enqueue  Q value ≜
  let node = ref (Some value, ref (None)) in
  (rec  loop_ =
    let tail = !(snd(! Q)) in
    let next = !(snd(! tail)) in
    if tail = !(snd(! Q)) then
      if next = None then
        if CAS (snd(! tail)) next node then
          CAS (snd(! Q)) tail node
        else loop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

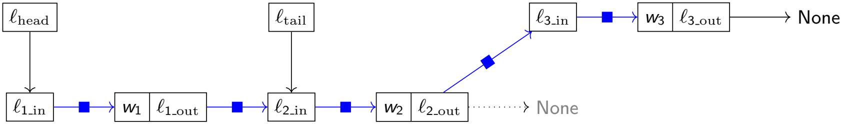
```
enqueue  Q value ≜
  let node = ref (Some value, ref (None)) in
  (rec  loop_ =
    let tail = !(snd(! Q)) in
    let next = !(snd(! tail)) in
    if tail = !(snd(! Q)) then
      if next = None then
        if CAS (snd(! tail)) next node then
          CAS (snd(! Q)) tail node
        else loop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

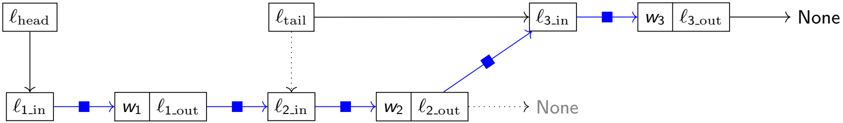
```
enqueue  Q value ≜
  let node = ref (Some value, ref (None)) in
  (rec  loop_ =
    let tail = !(snd(! Q)) in
    let next = !(snd(! tail)) in
    if tail = !(snd(! Q)) then
      if next = None then
        if CAS (snd(! tail)) next node then
          CAS (snd(! Q)) tail node
        else loop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

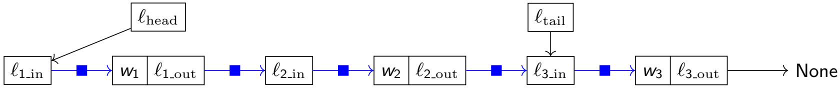
```
enqueue  Q value ≜
  let node = ref (Some value, ref (None)) in
  (rec  loop_ =
    let tail = !(snd(! Q)) in
    let next = !(snd(! tail)) in
    if tail = !(snd(! Q)) then
      if next = None then
        if CAS (snd(! tail)) next node then
          CAS (snd(! Q)) tail node
        else loop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

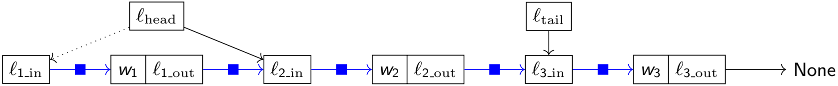
```
dequeue Q ≜
  (rec loop_ =
    let head = !(fst(! Q)) in
    let tail = !(snd(! Q)) in
    let p = newproph in
    let next = !(snd(! head)) in
    if head = Resolve(!(fst(! Q)), p, ()) then
      if head = tail then
        if next = None then
          None
        else
          CAS(snd(! Q))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

►**create slide**◄

In Coq!