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 safety of the two M&S Queues
- The project later generalised the results to apply to queues in general

Overview of the Project and Contributions

- Initial goal was to prove safety 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 safety of concurrent queues
    - Useful for *some* concurrent clients
    - Doesn't track queue contents
  - HOCAP-style specification
    - Stronger specification, useful for more complex clients
    - Tracks queue contents
    - Demonstrated with a PoC queue client, QueueAdd

Overview of the Project and Contributions

- Initial goal was to prove safety 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 safety of concurrent queues
    - Useful for *some* concurrent clients
    - Doesn't track queue contents
  - HOCAP-style specification
    - Stronger specification, useful for more complex clients
    - Tracks queue contents
    - Demonstrated with a PoC queue client, QueueAdd
- It was demonstrated that the HOCAP-style specification derives the sequential and concurrent specifications

Overview of the Project and Contributions

- Initial goal was to prove safety 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 safety of concurrent queues
    - Useful for *some* concurrent clients
    - Doesn't track queue contents
  - HOCAP-style specification
    - Stronger specification, useful for more complex clients
    - Tracks queue contents
    - Demonstrated with a PoC queue client, QueueAdd
- It was demonstrated that the HOCAP-style specification derives the sequential and concurrent specifications
- Implementations of the M&S Queues in HeapLang were proven to meet the three specifications
  - In particular, both version are safe

Overview of the Project and Contributions

- Initial goal was to prove safety 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 safety of concurrent queues
    - Useful for *some* concurrent clients
    - Doesn't track queue contents
  - HOCAP-style specification
    - Stronger specification, useful for more complex clients
    - Tracks queue contents
    - Demonstrated with a PoC queue client, QueueAdd
- It was demonstrated that the HOCAP-style specification derives the sequential and concurrent specifications
- Implementations of the M&S Queues in HeapLang were proven to meet the three specifications
  - In particular, both version are safe
- All proofs have been mechanised in the Coq proof assistant

Outline

1 Queue Specifications

2 The Two-Lock Michael-Scott Queue

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

4 The Lock-Free Michael-Scott Queue

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

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 value v at end of queue and return Some v
  - If empty, [], return None

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, $xs_v \mathrel{++} v$, remove value v at end of queue 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

HOCAP-style Specification - Abstract State RA

- We will need a construction to allow clients to track contents of queue

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

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 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)

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

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

HOCAP-style Specification

- Post-condition of initialize specification 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)

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

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

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

HOCAP-style Specification

- Post-condition of initialize specification 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)

$$
\begin{array}{ll}
\multicolumn{2}{l}{\exists\,\mathsf{isQueue} : \mathit{Val} \rightarrow \mathit{Qgnames} \rightarrow \mathsf{Prop}.} \\
& \forall v_q, G.\ \mathsf{isQueue}(v_q, G) \implies \Box\,\mathsf{isQueue}(v_q, G) \\
\wedge & \{\mathsf{True}\}\ \mathsf{initialize}\ ()\ \{v_q.\exists G.\ \mathsf{isQueue}(v_q, G) * G.\gamma_{\mathrm{Abst}} \Mapsto_\circ []\} \\
\wedge & \forall v_q, v, G, P, Q.\ \ \Big(\forall xs_v.\ G.\gamma_{\mathrm{Abst}} \Mapsto_\bullet xs_v * P \Rrightarrow_{\mathcal{E}\setminus\mathcal{N}.i\uparrow} \triangleright G.\gamma_{\mathrm{Abst}} \Mapsto_\bullet (v :: xs_v) * Q\Big) \mathrel{-\!\!*} \\
& \qquad \{\mathsf{isQueue}(v_q, G) * P\}\ \mathsf{enqueue}\ \ v_q\ \ v\ \{w.Q\}
\end{array}
$$

HOCAP-style Specification

- Post-condition of initialize specification 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)

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

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

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

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

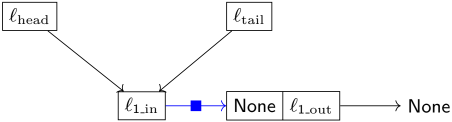
$$\land \quad \forall v_q, G, P, Q.\ \left(\forall xs_v.\ G.\gamma_{\mathrm{Abst}} \Mapsto_\bullet xs_v * P \Rrightarrow_{\mathcal{E}\setminus\mathcal{N}.i\uparrow} \triangleright \left(\begin{array}{l}(xs_v = [] * G.\gamma_{\mathrm{Abst}} \Mapsto_\bullet xs_v * Q(\mathrm{None})) \\ \lor \left(\begin{array}{l}\exists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

The Two-Lock Michael-Scott Queue

Implementation: initialize

- The queue data structure is a linked list

initialize $\triangleq$
  let *node* = ref (None, ref (None)) in
  let *H_lock* = newLock() in
  let *T_lock* = newLock() in
  ref ((ref (*node*), ref (*node*)), (*H_lock*, *T_lock*))

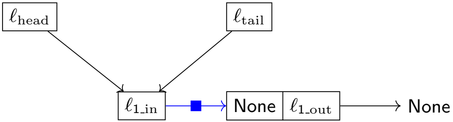


Implementation: initialize

- The queue 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 $\ell_{\text{in}} \mapsto (w, \ell_{\text{out}})$
- We use the following notation for nodes

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

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



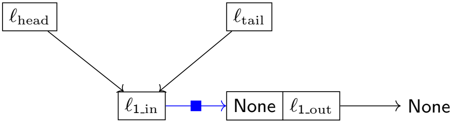
Implementation: initialize

- The queue 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 $\ell_{\text{in}} \mapsto (w, \ell_{\text{out}})$
- 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}

initialize $\triangleq$
  let $node$ = ref (None, ref (None)) in
  let H_lock = newLock() in
  let T_lock = newLock() in
  ref ((ref ($node$), ref ($node$)), (H_lock, T_lock))

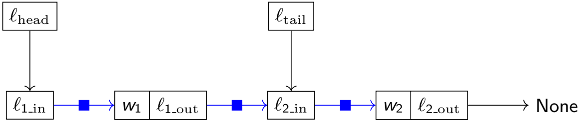


Implementation: enqueue

- 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

---

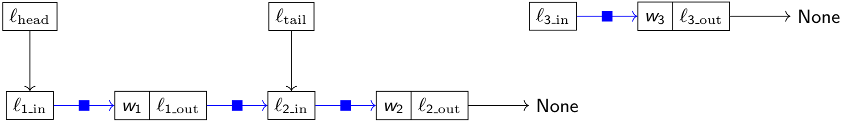
enqueue Q *value* $\triangleq$
  let *node* = ref (Some *value*, ref (None)) in
  acquire(snd(snd(! Q)));
  snd(!(!(snd(fst(! Q))))) $\leftarrow$ *node*;
  snd(fst(! Q)) $\leftarrow$ *node*;
  release(snd(snd(! Q)))



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

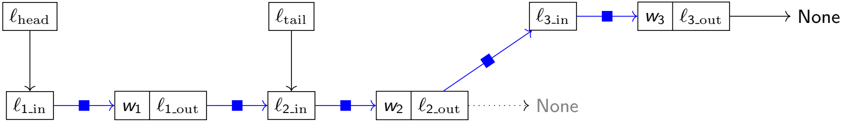
```
enqueue  Q value ≜
  let node = ref (Some value, ref (None)) in
  acquire(snd(snd(! Q)));
  snd(!(!(snd(fst(! Q))))) ← node;
  snd(fst(! Q)) ← node;
  release(snd(snd(! Q)))
```



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

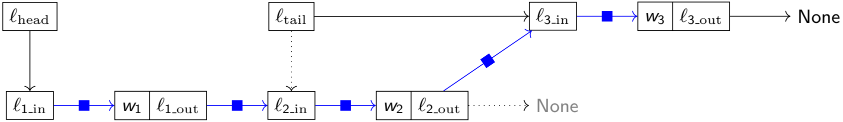
```
enqueue  Q value ≜
  let node = ref (Some value, ref (None)) in
  acquire(snd(snd(! Q)));
  snd(!(!(snd(fst(! Q))))) ← node;
  snd(fst(! Q)) ← node;
  release(snd(snd(! Q)))
```



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

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

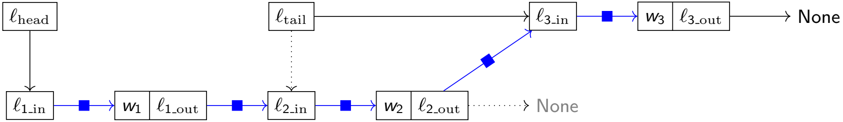


Implementation: enqueue

- 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
- Once a node is enqueued, its position in the linked list is fixed

---

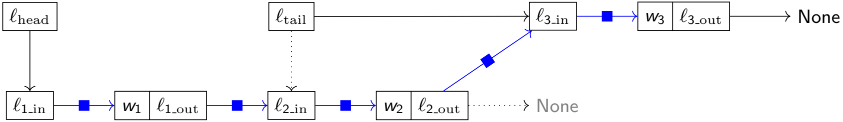
enqueue Q $value$ $\triangleq$
  let $node$ = ref (Some $value$, ref (None)) in
  acquire(snd(snd(! Q)));
  snd(!(!(snd(fst(! Q))))) $\leftarrow$ $node$;
  snd(fst(! Q)) $\leftarrow$ $node$;
  release(snd(snd(! Q)))



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
- Once a node is enqueued, its position in the linked list is fixed
- Adding and swinging not atomic $\rightarrow$ Tail node is either last or second last

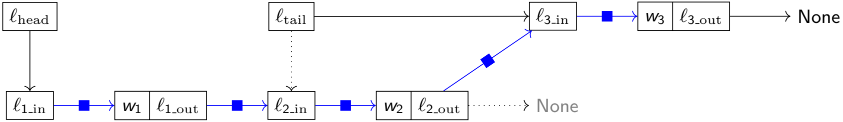
```
enqueue  Q value ≜
  let node = ref (Some value, ref (None)) in
  acquire(snd(snd(! Q)));
  snd(!(!(snd(fst(! Q))))) ← node;
  snd(fst(! Q)) ← node;
  release(snd(snd(! Q)))
```



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
- Once a node is enqueued, its position in the linked list is fixed
- Adding and swinging not atomic $\rightarrow$ Tail node is either last or second last
- dequeue ignores tail pointer $\rightarrow$ Tail node can lag behind head node

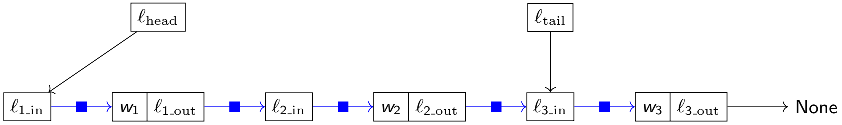
```
enqueue  Q value ≜
   let node = ref (Some value, ref (None)) in
   acquire(snd(snd(! Q)));
   snd(!(!(snd(fst(! Q))))) ← node;
   snd(fst(! Q)) ← node;
   release(snd(snd(! Q)))
```



Implementation: dequeue

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

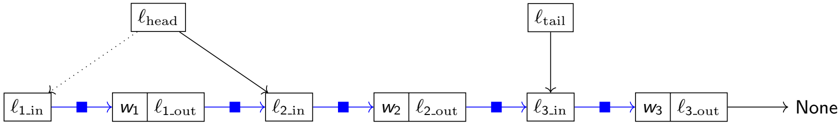
```
dequeue  Q ≜
  acquire(fst(snd(! Q)));
  let node = !(fst(fst(! Q))) in
  let new_head = !(snd(! node)) in
  if new_head = None then
    release(fst(snd(! Q)));
    None
  else
    let value = fst(! new_head) in
    fst(fst(! Q)) ← new_head;
    release(fst(snd(! Q)));
    value
```



Implementation: dequeue

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

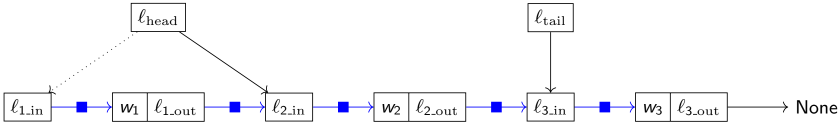
```
dequeue  Q ≜
  acquire(fst(snd(! Q)));
  let node = !(fst(fst(! Q))) in
  let new_head = !(snd(! node)) in
  if new_head = None then
    release(fst(snd(! Q)));
    None
  else
    let value = fst(! new_head) in
    fst(fst(! Q)) ← new_head;
    release(fst(snd(! Q)));
    value
```



Implementation: dequeue

- The dequeue function checks if queue is empty
  - If empty, return None
  - Else, swing head pointer to new head node, and return its value
- Dequeued node not freed → Linked list only grows

```
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
```



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

The isLL Predicate

- 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 Predicate)

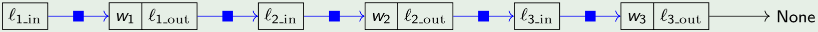
$$\begin{aligned}
\text{isLL_chain}([]) &\triangleq \text{True} \\
\text{isLL_chain}([x]) &\triangleq \text{in}(x) \mapsto^{\square} (\text{val}(x), \text{out}(x)) \\
\text{isLL_chain}(x :: x' :: xs) &\triangleq \text{in}(x) \mapsto^{\square} (\text{val}(x), \text{out}(x)) * \text{out}(x') \mapsto^{\square}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

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

Example

Consider the list: $xs = [(\ell_{3\_\text{in}}, w_3, \ell_{3\_\text{out}}); (\ell_{2\_\text{in}}, w_2, \ell_{2\_\text{out}}); (\ell_{1\_\text{in}}, w_1, \ell_{1\_\text{out}})]$.

$$\begin{aligned}
\text{isLL}(xs) = \; & \ell_{3_\text{out}} \mapsto \text{None} * \ell_{3_\text{in}} \mapsto^{\square} (w_3, \ell_{3_\text{out}}) * \ell_{2_\text{out}} \mapsto^{\square} \ell_{3_\text{in}} * \\
& \ell_{2_\text{in}} \mapsto^{\square} (w_2, \ell_{2_\text{out}}) * \ell_{1_\text{out}} \mapsto^{\square} \ell_{2_\text{in}} * \\
& \ell_{1_\text{in}} \mapsto^{\square} (w_1, \ell_{1_\text{out}})
\end{aligned}$$



Invariant

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

Invariant

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

---

- Contains abstract state of queue – existentially quantified as it can change

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

$$\mathsf{I}_{\mathsf{TLH}}(\ell_{\mathrm{head}}, \ell_{\mathrm{tail}}, G) \triangleq \exists xs_v.\ G.\gamma_{\mathrm{Abst}} \Mapsto_\bullet xs_v \ast \qquad \text{(abstract state)}$$

Invariant

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

---

- Contains abstract state of queue – existentially quantified as it can change
- Defines structure of the concrete linked list, *xsc*

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

$$\begin{aligned} I_{\mathrm{TLH}}(\ell_{\mathrm{head}}, \ell_{\mathrm{tail}}, G) \triangleq\ & \exists xs_v.\ G.\gamma_{\mathrm{Abst}} \Mapsto_\bullet xs_v * && \text{(abstract state)} \\ & \exists xs, xs_{\mathrm{queue}}, xs_{\mathrm{old}}, x_{\mathrm{head}}, x_{\mathrm{tail}}. && \text{(concrete state)} \\ & xs = xs_{\mathrm{queue}} +\!\!+ [x_{\mathrm{head}}] +\!\!+ xs_{\mathrm{old}} * \\ & \mathsf{isLL}(xs) * \end{aligned}$$

Invariant

- Queue predicate must be persistent (according to specification)
- Problem: the queue relies on non-persistent resources (e.g. $\ell_{\text{head}} \mapsto \text{in}(x_{\text{head}})$)
- Solution: identify an *invariant* (persistent), describing the resources

---

- Contains abstract state of queue – existentially quantified as it can change
- Defines structure of the concrete linked list, *xsc*
- Asserts relation between abstract state and concrete state

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

$$
\begin{aligned}
\mathsf{I}_{\mathsf{TLH}}(\ell_{\text{head}}, \ell_{\text{tail}}, G) \triangleq\ & \exists xs_v.\ G.\gamma_{\text{Abst}} \Mapsto_\bullet xs_v \ast & \text{(abstract state)}\\
& \exists xs, xs_{\text{queue}}, xs_{\text{old}}, x_{\text{head}}, x_{\text{tail}}. & \text{(concrete state)}\\
& xs = xs_{\text{queue}} +\!\!+ [x_{\text{head}}] +\!\!+ xs_{\text{old}} \ast &\\
& \text{isLL}(xs) \ast &\\
& \text{projVal}(xs_{\text{queue}}) = \text{wrapSome}(xs_v) \ast &
\end{aligned}
$$

Invariant

- Queue predicate must be persistent (according to specification)
- Problem: the queue relies on non-persistent resources (e.g. $\ell_{\text{head}} \mapsto \text{in}(x_{\text{head}})$)
- Solution: identify an *invariant* (persistent), describing the resources

---

- Contains abstract state of queue – existentially quantified as it can change
- Defines structure of the concrete linked list, *xsc*
- Asserts relation between abstract state and concrete state
- Identifies possible queue states: **Static**, **Enqueue**, **Dequeue**, and **Both**
  - Two locks → Four queue states
  - Invariant describes the queue resources in each state
  - See next slide

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

$$
\begin{aligned}
\mathrm{I}_{\mathsf{TLH}}(\ell_{\text{head}}, \ell_{\text{tail}}, G) \triangleq\ & \exists xs_v.\ G.\gamma_{\text{Abst}} \Mapsto_{\bullet} xs_v * && \text{(abstract state)} \\
& \exists xs, xs_{\text{queue}}, xs_{\text{old}}, x_{\text{head}}, x_{\text{tail}}. && \text{(concrete state)} \\
& xs = xs_{\text{queue}} +\!\!+ [x_{\text{head}}] +\!\!+ xs_{\text{old}} * \\
& \mathsf{isLL}(xs) * \\
& \mathsf{projVal}(xs_{\text{queue}}) = \mathsf{wrapSome}(xs_v) * \\
& \ldots
\end{aligned}
$$

Invariant (Queue States)

- Idea: the enqueueing thread keeps half of tail pointer between invariant openings
- Guarantees that the pointer is not updated (full pointer needed for update)
- Similarly for the dequeueing thread
- **Enqueue** and **Both** also captures "gap" between adding x_{new} and swinging ℓ_{tail}
- Tokens used to reason about which state queue is in

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

...

$$\ell_{\mathrm{head}} \mapsto \mathrm{in}(x_{\mathrm{head}}) * \ell_{\mathrm{tail}} \mapsto \mathrm{in}(x_{\mathrm{tail}}) * \mathrm{isLast}(x_{\mathrm{tail}}, xs) * \mathrm{TokNE}\ G * \mathrm{TokND}\ G * \mathrm{TokUpdated}\ G \qquad \text{(Static)}$$

$$\vee\ \ell_{\mathrm{head}} \mapsto \mathrm{in}(x_{\mathrm{head}}) * \ell_{\mathrm{tail}} \mapsto^{\frac{1}{2}} \mathrm{in}(x_{\mathrm{tail}}) * (\mathrm{isLast}(x_{\mathrm{tail}}, xs) * \mathrm{TokBefore}\ G \vee \mathrm{isSndLast}(x_{\mathrm{tail}}, xs) * \mathrm{TokAfter}\ G) * \mathrm{TokE}\ G * \mathrm{TokND}\ G \qquad \text{(Enqueue)}$$

$$\vee\ \ell_{\mathrm{head}} \mapsto^{\frac{1}{2}} \mathrm{in}(x_{\mathrm{head}}) * \ell_{\mathrm{tail}} \mapsto \mathrm{in}(x_{\mathrm{tail}}) * \mathrm{isLast}(x_{\mathrm{tail}}, xs) * \mathrm{TokNE}\ G * \mathrm{TokD}\ G * \mathrm{TokUpdated}\ G \qquad \text{(Dequeue)}$$

$$\vee\ \ell_{\mathrm{head}} \mapsto^{\frac{1}{2}} \mathrm{in}(x_{\mathrm{head}}) * \ell_{\mathrm{tail}} \mapsto^{\frac{1}{2}} \mathrm{in}(x_{\mathrm{tail}}) * (\mathrm{isLast}(x_{\mathrm{tail}}, xs) * \mathrm{TokBefore}\ G \vee \mathrm{isSndLast}(x_{\mathrm{tail}}, xs) * \mathrm{TokAfter}\ G) * \mathrm{TokE}\ G * \mathrm{TokD}\ G \qquad \text{(Both)}$$

Queue Predicate

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

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)

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

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)

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

- The queue predicate is persistent, as all its constituents are

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)

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

- The queue predicate is persistent, as all its constituents are
- Proving that Two-Lock M&S Queue 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

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

Proof Sketch of the Hoare Triple for enqueue

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

(Proof)

Assume the view-shift, and the persistent information in $\mathrm{isQueue}(v_q, Qgnames)$:

- $v_q = \ell_{\mathrm{queue}} * \ell_{\mathrm{queue}} \mapsto^{\square} ((\ell_{\mathrm{head}}, \ell_{\mathrm{tail}}), (h_{\mathrm{lock}}, t_{\mathrm{lock}}))$
- $\boxed{\mathrm{I}_{\mathrm{TLH}}(\ell_{\mathrm{head}}, \ell_{\mathrm{tail}}, G)}^{\mathcal{N}.queue}$
- $\mathrm{isLock}(G.\gamma_{\mathrm{Tlock}}, t_{\mathrm{lock}}, \mathrm{TokE}\ \ G)$

Proof Sketch of the Hoare Triple for enqueue

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

(Proof)

Assume the view-shift, and the persistent information in $\text{isQueue}(v_q, Qgnames)$:

- $v_q = \ell_{\text{queue}} * \ell_{\text{queue}} \mapsto^{\square} ((\ell_{\text{head}}, \ell_{\text{tail}}), (h_{\text{lock}}, t_{\text{lock}}))$
- $\boxed{\mathrm{I}_{\mathrm{TLH}}(\ell_{\text{head}}, \ell_{\text{tail}}, G)}^{\mathcal{N}.queue}$
- $\text{isLock}(G.\gamma_{\text{Tlock}}, t_{\text{lock}}, \text{TokE}\ \ G)$

$\{P\}$
```
let node = ref (Some v, ref (None)) in

acquire(snd(snd(! v_q)));

e_t = !(snd(fst(! v_q)))

snd(!(e_t)) ← node;

snd(fst(! v_q)) ← node;

release(snd(snd(! v_q)))
```
$\{Q\}$

Proof Sketch of the Hoare Triple for enqueue

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

(Proof)

Assume the view-shift, and the persistent information in $\mathrm{isQueue}(v_q, Qgnames)$:

- $v_q = \ell_{\mathrm{queue}} * \ell_{\mathrm{queue}} \mapsto^{\square} ((\ell_{\mathrm{head}}, \ell_{\mathrm{tail}}), (h_{\mathrm{lock}}, t_{\mathrm{lock}}))$
- $\boxed{\mathrm{I}_{\mathrm{TLH}}(\ell_{\mathrm{head}}, \ell_{\mathrm{tail}}, G)}^{\mathcal{N}.queue}$
- $\mathrm{isLock}(G.\gamma_{\mathrm{Tlock}}, t_{\mathrm{lock}}, \mathrm{TokE}\ \ G)$

$\{P\}$

  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)));

  e_t = !(snd(fst(! v_q)))

  snd(!(e_t)) ← $node$;

  snd(fst(! v_q)) ← $node$;

  release(snd(snd(! v_q)))

$\{Q\}$

Proof Sketch of the Hoare Triple for enqueue

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

(Proof)

Assume the view-shift, and the persistent information in $\mathsf{isQueue}(v_q, Qgnames)$:

- $v_q = \ell_{\mathrm{queue}} * \ell_{\mathrm{queue}} \mapsto^{\Box} ((\ell_{\mathrm{head}}, \ell_{\mathrm{tail}}), (h_{\mathrm{lock}}, t_{\mathrm{lock}}))$
- $\boxed{\mathsf{I}_{\mathsf{TLH}}(\ell_{\mathrm{head}}, \ell_{\mathrm{tail}}, G)}^{\mathcal{N}.queue}$
- $\mathsf{isLock}(G.\gamma_{\mathrm{Tlock}}, t_{\mathrm{lock}}, \mathsf{TokE}\ \ G)$

$\{P\}$
  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)))

  snd(!(e_t)) ← $node$;

  snd(fst(! v_q)) ← $node$;

  release(snd(snd(! v_q)))
$\{Q\}$

Proof Sketch of the Hoare Triple for enqueue

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

(Proof)

Assume the view-shift, and the persistent information in isQueue(v_q, *Qgnames*):

- $v_q = \ell_{\text{queue}} * \ell_{\text{queue}} \mapsto^{\square} ((\ell_{\text{head}}, \ell_{\text{tail}}), (h_{\text{lock}}, t_{\text{lock}}))$
- $\boxed{\mathrm{I}_{\text{TLH}}(\ell_{\text{head}}, \ell_{\text{tail}}, G)}^{\mathcal{N}.queue}$
- $\text{isLock}(G.\gamma_{\text{Tlock}}, t_{\text{lock}}, \text{TokE}\ \ G)$

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

  snd(fst(! v_q)) ← *node*;

  release(snd(snd(! v_q)))
$\{Q\}$

Proof Sketch of the Hoare Triple for enqueue

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

(Proof)

Assume the view-shift, and the persistent information in $\mathsf{isQueue}(v_q, Qgnames)$:

- $v_q = \ell_{\mathrm{queue}} * \ell_{\mathrm{queue}} \mapsto^{\Box} ((\ell_{\mathrm{head}}, \ell_{\mathrm{tail}}), (h_{\mathrm{lock}}, t_{\mathrm{lock}}))$
- $\boxed{\mathsf{I_{TLH}}(\ell_{\mathrm{head}}, \ell_{\mathrm{tail}}, G)}^{\mathcal{N}.queue}$
- $\mathsf{isLock}(G.\gamma_{\mathrm{Tlock}}, t_{\mathrm{lock}}, \mathsf{TokE}\ \ G)$

$\{P\}$

  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}. $\mathsf{I_{TLH}}$: **Static**/**Dequeue** $\to$ **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)) $\leftarrow$ $node$;   (make x_{tail} point to x_{new}. $\mathsf{I_{TLH}}$: **Enqueue**/**Both** (before) $\to$ **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)) $\leftarrow$ $node$;

  release(snd(snd(! v_q)))

$\{Q\}$

Proof Sketch of the Hoare Triple for enqueue

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

(Proof)

Assume the view-shift, and the persistent information in $\mathrm{isQueue}(v_q, Qgnames)$:

- $v_q = \ell_{\mathrm{queue}} * \ell_{\mathrm{queue}} \mapsto^{\square} ((\ell_{\mathrm{head}}, \ell_{\mathrm{tail}}), (h_{\mathrm{lock}}, t_{\mathrm{lock}}))$
- $\boxed{\mathrm{I_{TLH}}(\ell_{\mathrm{head}}, \ell_{\mathrm{tail}}, G)}^{\mathcal{N}.queue}$
- $\mathrm{isLock}(G.\gamma_{\mathrm{Tlock}}, t_{\mathrm{lock}}, \mathrm{TokE}\ \ G)$

$\{P\}$

  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)))

$\{Q\}$

Proof Sketch of the Hoare Triple for enqueue

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

(Proof)

Assume the view-shift, and the persistent information in $\text{isQueue}(v_q, Qgnames)$:

- $v_q = \ell_{\text{queue}} * \ell_{\text{queue}} \mapsto^{\square} ((\ell_{\text{head}}, \ell_{\text{tail}}), (h_{\text{lock}}, t_{\text{lock}}))$
- $\boxed{\text{I}_{\text{TLH}}(\ell_{\text{head}}, \ell_{\text{tail}}, G)}^{\mathcal{N}.queue}$
- $\text{isLock}(G.\gamma_{\text{Tlock}}, t_{\text{lock}}, \text{TokE}\ \ G)$

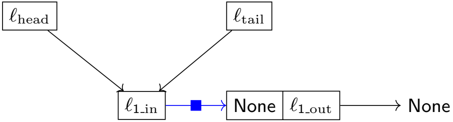
$\{P\}$
  let $node$ = ref (Some v, ref (None)) in   (create node x_{new})
$\{P * \ \text{out}(x_{\text{new}}) \mapsto \text{None}\}$
  acquire(snd(snd(! v_q)));   (acquire tail lock)
$\{P * \ \text{out}(x_{\text{new}}) \mapsto \text{None} * \text{TokE}\ \ G\}$
  e_t = !(snd(fst(! v_q)))   (find current tail, x_{tail}. I_{TLH}: **Static/Dequeue** → **Enqueue/Both** (before))
$\{P * \ \text{out}(x_{\text{new}}) \mapsto \text{None} * \ell_{\text{tail}} \mapsto^{\frac{1}{2}} \text{in}(x_{\text{tail}}) * \ \text{TokNE}\ \ G * \ \text{TokAfter}\ \ G\}$
  snd(!(e_t)) ← $node$;   (make x_{tail} point to x_{new}. I_{TLH}: **Enqueue/Both** (before) → **Enqueue/Both** (after))
$\{Q * \ell_{\text{tail}} \mapsto^{\frac{1}{2}} \text{in}(x_{\text{tail}}) * \ \text{TokNE}\ \ G * \ \text{TokBefore}\ \ G\}$
  snd(fst(! v_q)) ← $node$;   (swing tail pointer to x_{new}. I_{TLH}: **Enqueue/Both** (after) → **Static/Dequeue**)
$\{Q * \ \text{TokE}\ \ G\}$
  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 – now with CAS instructions
- No longer need locks

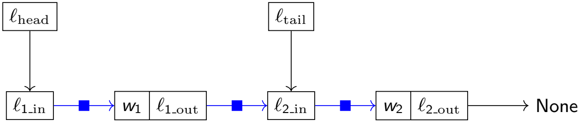
initialize $\triangleq$
  let *node* = ref (None, ref (None)) in
  ref (ref (*node*), ref (*node*))



Implementation: enqueue

- Appending x_{new} to linked list is now done with CAS
- Ensures that no other thread has performed an enqueue during own enqueue
  - Otherwise, we might "overwrite" another threads enqueued node
- Swinging tail to x_{new} might fail – another thread has helped us

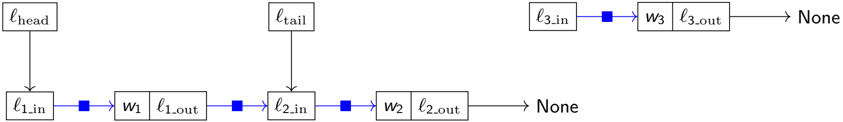
```
enqueue  Q value ≜
  let node = ref (Some value, ref (None)) in
  (rec  loop_ =
    let tail = !(snd(! Q)) in
    let next = !(snd(! tail)) in
    if tail = !(snd(! Q)) then
      if next = None then
        if CAS (snd(! tail)) next node then
          CAS (snd(! Q)) tail node
        else loop ()
      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 during own enqueue
  - Otherwise, we might “overwrite” another threads enqueued node
- Swinging tail to x_{new} might fail – another thread has helped us

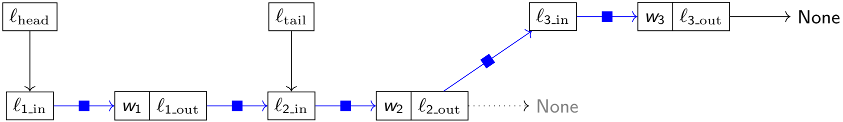
```
enqueue  Q value ≜
  let node = ref (Some value, ref (None)) in
  (rec  loop_ =
    let tail = !(snd(! Q)) in
    let next = !(snd(! tail)) in
    if tail = !(snd(! Q)) then
      if next = None then
        if CAS (snd(! tail)) next node then
          CAS (snd(! Q)) tail node
         else loop ()
       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 during own enqueue
  - Otherwise, we might "overwrite" another threads enqueued node
- Swinging tail to x_{new} might fail – another thread has helped us

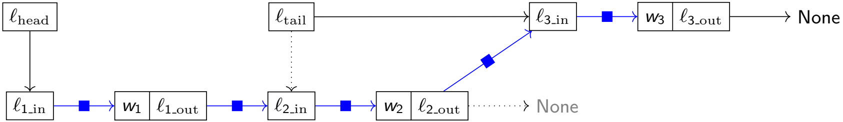
```
enqueue  Q value ≜
  let node = ref (Some value, ref (None)) in
  (rec  loop_ =
    let tail = !(snd(! Q)) in
    let next = !(snd(! tail)) in
    if tail = !(snd(! Q)) then
      if next = None then
        if CAS (snd(! tail)) next node then
          CAS (snd(! Q)) tail node
        else loop ()
      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 during own enqueue
  - Otherwise, we might "overwrite" another threads enqueued node
- Swinging tail to x_{new} might fail – another thread has helped us

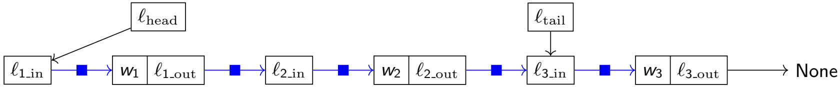
```
enqueue  Q value ≜
  let node = ref (Some value, ref (None)) in
  (rec  loop_ =
    let tail = !(snd(! Q)) in
    let next = !(snd(! tail)) in
    if tail = !(snd(! Q)) then
      if next = None then
        if CAS (snd(! tail)) next node then
          CAS (snd(! Q)) tail node
        else loop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



Implementation: dequeue

- Head now swung with CAS instruction
- Ensures that no other thread has dequeued the element we are trying to dequeue

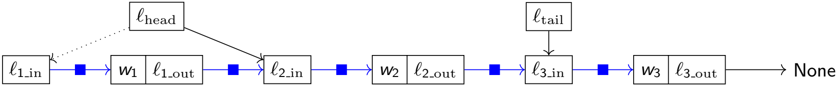
```
dequeue Q ≜
  (rec loop_ =
    let head = !(fst(! Q)) in
    let tail = !(snd(! Q)) in
    let p = newproph in
    let next = !(snd(! head)) in
    if head = Resolve(!(fst(! Q)), p, ()) then
      if head = tail then
        if next = None then
          None
        else
          CAS(snd(! Q)) 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 instruction
- Ensures that no other thread has 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

- Proving adherence to HOCAP-style specification requires applying the view-shift at some point (must update P to Q)

```
...
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 loop ()
...
```

Prophecies

- Proving adherence to HOCAP-style specification requires applying the view-shift at some point (must update P to Q)
- View-shift is applied at *Linearisation Points* – points where the effect of the function takes place

```
...
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 loop ()
...
```

Prophecies

- Proving adherence to HOCAP-style specification requires applying the view-shift at some point (must update P to Q)
- View-shift is applied at *Linearisation Points* – points where the effect of the function takes place
- When the queue is empty, the linearisation point is when reading *next* (specifically, the dereference instruction)
- We deduce that at exactly that read, the queue was empty

```
...
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 loop ()
...
```

Prophecies

- Proving adherence to HOCAP-style specification requires applying the view-shift at some point (must update P to Q)
- View-shift is applied at *Linearisation Points* – points where the effect of the function takes place
- When the queue is empty, the linearisation point is when reading *next* (specifically, the dereference instruction)
- We deduce that at exactly that read, the queue was empty
- But we only conclude the queue is empty if consistency check on next line succeeds
- The dereference is only the linearisation point if consistency check succeeds

```
...
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 loop ()
...
```

Prophecies

- Proving adherence to HOCAP-style specification requires applying the view-shift at some point (must update P to Q)
- View-shift is applied at *Linearisation Points* – points where the effect of the function takes place
- When the queue is empty, the linearisation point is when reading *next* (specifically, the dereference instruction)
- We deduce that at exactly that read, the queue was empty
- But we only conclude the queue is empty if consistency check on next line succeeds
- The dereference is only the linearisation point if consistency check succeeds
- Prophecies: reason about future computations (e.g. the consistency check)
  - $!(\mathsf{fst}(!\,Q))$ will evaluate to some v_p (later proof obligation)
  - Before reading *next*, reason about whether $head = v_p$

```
...
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 loop ()
...
```

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

- Reason for consistency checks: ABA problem in original implementation

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

- Reason for consistency checks: ABA problem in original implementation
- HeapLang is garbage collected language, so we can remove consistency checks
- 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 ()
      )()
```

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

- Reason for consistency checks: ABA problem in original implementation
- HeapLang is garbage collected language, so we can remove consistency checks
- 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
- Correctness: both versions shown to satisfy HOCAP-style specification...

```
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)

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

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)

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

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

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

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

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 $\textsc{Auth}(\mathcal{P}(\mathit{Node}))$

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, $\gamma_{\text{Tail}} \rightarrowtail x_n$ means that the current tail node is x_n
- And $x_m \dashrightarrow \gamma_{\text{Tail}}$ means that node x_m can always reach the tail node

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 $\text{AUTH}(\mathcal{P}(\mathit{Node}))$
- Defined in two parts: Abstract Points-to ($\gamma \rightarrowtail x$) and Abstract Reach ($x \dashrightarrow \gamma$)
- For instance, $\gamma_{\text{Tail}} \rightarrowtail x_n$ means that the current tail node is x_n
- And $x_m \dashrightarrow \gamma_{\text{Tail}}$ means that node x_m can always reach the tail node

Lemmas for Reachability (simplified)

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

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

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

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

In Coq!