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

Queue Specifications (HOCAP)

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, $xs_v ++ v$, 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

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

$\gamma \Mapsto_{\bullet} xs_v$

Owned by queue

Fragmental view

$\gamma \Mapsto_{\circ} xs_v$

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**
$\gamma \Rrightarrow_{\bullet} xs_v$	$\gamma \Rrightarrow_{\circ} xs_v$
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

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

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

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

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)

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

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

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)

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

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

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

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)

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

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

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

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

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)

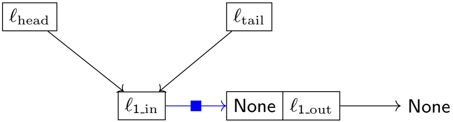
$$
\begin{aligned}
&\exists\, \mathsf{isQueue} : Val \to Qgnames \to \mathsf{Prop}.\\
&\quad \forall v_q, G.\ \mathsf{isQueue}(v_q, G) \implies \Box\, \mathsf{isQueue}(v_q, G)\\
\land\quad & \{\mathsf{True}\}\ \mathsf{initialize}\ ()\ \{v_q.\exists G.\ \mathsf{isQueue}(v_q, G) * G.\gamma_{\mathrm{Abst}}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

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*))

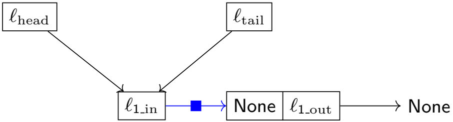


Implementation: initialize

- 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

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

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*))



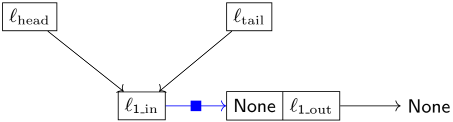
Implementation: initialize

- 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

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

- 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 ≜
  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))
```

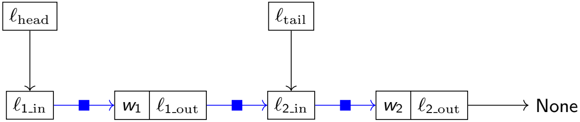


Implementation: enqueue

- 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

---

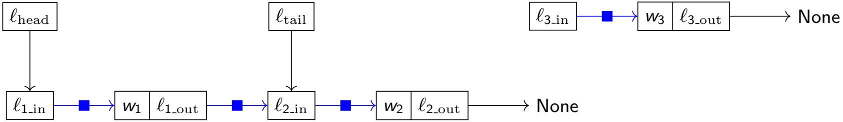
enqueue Q *value* $\triangleq$
  let *node* = ref (Some *value*, ref (None)) in
  acquire(snd(snd(! Q)));
  snd(!(!(snd(fst(! Q))))) $\leftarrow$ *node*;
  snd(fst(! Q)) $\leftarrow$ *node*;
  release(snd(snd(! Q)))



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

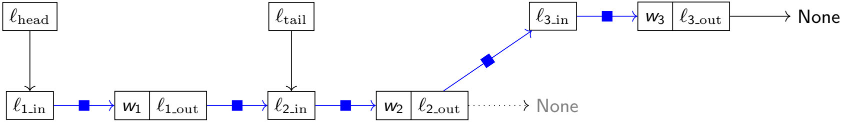
enqueue Q *value* $\triangleq$
  let *node* = ref (Some *value*, ref (None)) in
  acquire(snd(snd(! Q)));
  snd(!(!(snd(fst(! Q))))) $\leftarrow$ *node*;
  snd(fst(! Q)) $\leftarrow$ *node*;
  release(snd(snd(! Q)))



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

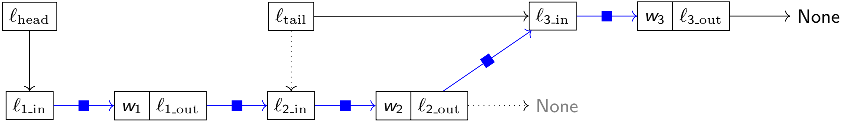
enqueue Q *value* $\triangleq$
  let *node* = ref (Some *value*, ref (None)) in
  acquire(snd(snd(! Q)));
  snd(!(!(snd(fst(! Q))))) $\leftarrow$ *node*;
  snd(fst(! Q)) $\leftarrow$ *node*;
  release(snd(snd(! Q)))



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

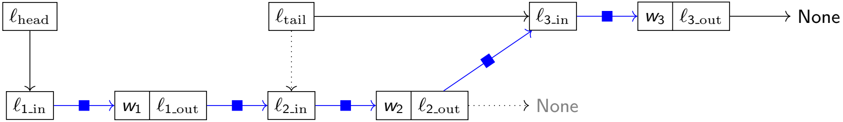
enqueue Q *value* $\triangleq$
  let *node* = ref (Some *value*, ref (None)) in
  acquire(snd(snd(! Q)));
  snd(!(!(snd(fst(! Q))))) $\leftarrow$ *node*;
  snd(fst(! Q)) $\leftarrow$ *node*;
  release(snd(snd(! Q)))



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

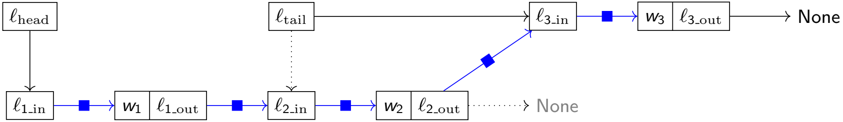
```
enqueue  Q value ≜
   let node = ref (Some value, ref (None)) in
   acquire(snd(snd(! Q)));
   snd(!(!(snd(fst(! Q))))) ← node;
   snd(fst(! Q)) ← node;
   release(snd(snd(! Q)))
```



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

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)))

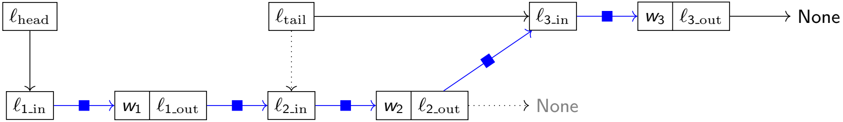


Implementation: enqueue

- 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
- dequeue ignores tail pointer $\rightarrow$ Tail node can lag behind head node

---

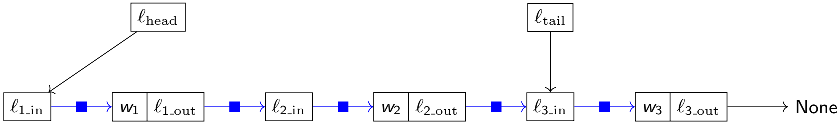
enqueue Q *value* $\triangleq$
let *node* = ref (Some *value*, ref (None)) in
acquire(snd(snd(! Q)));
snd(!(!(snd(fst(! Q))))) $\leftarrow$ *node*;
snd(fst(! Q)) $\leftarrow$ *node*;
release(snd(snd(! Q)))



Implementation: 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

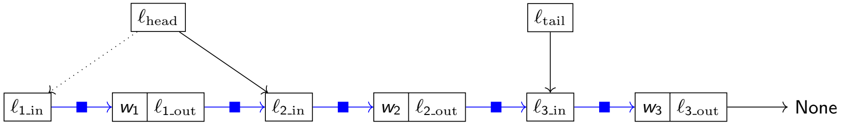
```
dequeue  Q ≜
  acquire(fst(snd(! Q)));
  let node = !(fst(fst(! Q))) in
  let new_head = !(snd(! node)) in
  if new_head = None then
    release(fst(snd(! Q)));
    None
  else
    let value = fst(! new_head) in
    fst(fst(! Q)) ← new_head;
    release(fst(snd(! Q)));
    value
```



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

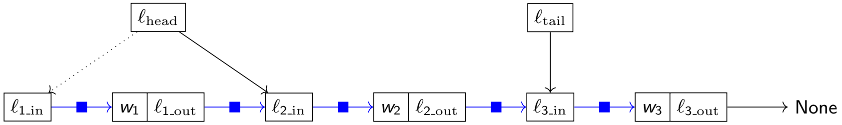
```
dequeue  Q ≜
   acquire(fst(snd(! Q)));
   let node = !(fst(fst(! Q))) in
   let new_head = !(snd(! node)) in
   if new_head = None then
      release(fst(snd(! Q)));
      None
    else
      let value = fst(! new_head) in
      fst(fst(! Q)) ← new_head;
      release(fst(snd(! Q)));
      value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



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 $\rightarrow$ 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)

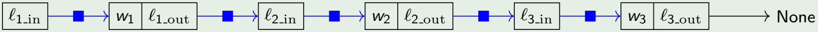
$$\begin{aligned}
\mathsf{isLL_chain}([]) &\triangleq \mathsf{True} \\
\mathsf{isLL_chain}([x]) &\triangleq \mathsf{in}(x) \mapsto^{\square} (\mathsf{val}(x), \mathsf{out}(x)) \\
\mathsf{isLL_chain}(x :: x' :: xs) &\triangleq \mathsf{in}(x) \mapsto^{\square} (\mathsf{val}(x), \mathsf{out}(x)) * \mathsf{out}(x')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

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

Example

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

$$\begin{aligned}
\mathsf{isLL}(xs) = \ \ell_{3_out} \mapsto \mathsf{None} \ * \ & \ell_{3_in} \mapsto^{\square} (w_3, \ell_{3_out}) * \ell_{2_out} \mapsto^{\square} \ell_{3_in} * \\
& \ell_{2_in} \mapsto^{\square} (w_2, \ell_{2_out}) * \ell_{1_out} \mapsto^{\square} \ell_{2_in} * \\
& \ell_{1_in} \mapsto^{\square} (w_1, \ell_{1_out})
\end{aligned}$$



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. $\ell_{\text{head}} \mapsto \text{in}(x_{\text{head}})$)
- Solution: identify an *invariant* (persistent), describing the resources

---

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

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

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

Invariant

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

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

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

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

---

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

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

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

Invariant

- Queue predicate must be persistent (according to specification)
- Problem: the queue relies on non-persistent resources (e.g. $\ell_{\mathrm{head}} \mapsto \mathsf{in}(x_{\mathrm{head}})$)
- 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, xs
- Asserts relation between abstract state and concrete state
- Identifies possible queue states: **Static**, **Enqueue**, **Dequeue**, and **Both**
  - Two locks $\rightarrow$ Four queue states
  - Invariant describes the queue resources in each state
  - See next slide

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

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

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)

...

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

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

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

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

Queue Predicate

- We now define the queue predicate in terms of our invariant

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

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

Queue Predicate

- We now define the queue predicate 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

- We now define the queue predicate 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 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

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

Proof Sketch of the Hoare Triple for enqueue

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

(Proof)

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

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

Proof Sketch of the Hoare Triple for enqueue

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

(Proof)

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

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

$\{P\}$

```
let node = ref (Some v, ref (None)) in

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

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

(Proof)

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

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

$\{P\}$

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

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

(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)))

  snd(!(e_t)) $\leftarrow$ $node$;

  snd(fst(! v_q)) $\leftarrow$ $node$;

  release(snd(snd(! v_q)))

$\{Q\}$

Proof Sketch of the Hoare Triple for enqueue

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

(Proof)

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

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

$\{P\}$

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))); (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}. $\mathrm{I}_{\mathrm{TLH}}$: **Static**/**Dequeue** $\to$ **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)) $\leftarrow$ $node$;

snd(fst(! v_q)) $\leftarrow$ $node$;

release(snd(snd(! v_q)))

$\{Q\}$

Proof Sketch of the Hoare Triple for enqueue

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

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

release(snd(snd(! v_q)))

$\{Q\}$

Proof Sketch of the Hoare Triple for enqueue

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

(Proof)

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

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

$\{P\}$

  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** $\rightarrow$ **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}. VS. $\mathsf{I}_{\mathsf{TLH}}$: **Enqueue**/**Both** (before) $\rightarrow$ **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)) $\leftarrow$ $node$;   (swing tail pointer to x_{new}. $\mathsf{I}_{\mathsf{TLH}}$: **Enqueue**/**Both** (after) $\rightarrow$ **Static**/**Dequeue**)

$\{Q * \mathsf{TokE}\ \ G\}$

  release(snd(snd(! v_q)))

$\{Q\}$

Proof Sketch of the Hoare Triple for enqueue

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

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

$\{Q * \mathrm{TokE}\ \ G\}$

release(snd(snd(! v_q))) (release tail lock)

$\{Q\}$

The Lock-Free Michael-Scott Queue

Implementation (Consistency-Check Free)

- 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

Implementation (Consistency-Check Free)

- 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
- Ensures no overwrites of enqueued nodes, and no node dequeued twice

```
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 ()
  )()
```

Consistency Checks and Prophecies

- Original implementation has consistency checks to deal with ABA problem
- Creates complications when proving adherence to HOCAP-style specification
- In dequeue, the point at which to apply the view-shift (to update P to Q) depends on whether queue is empty and result of future consistency check
- Prophecies allow us to reason about future computations (e.g. consistency check)

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

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

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

Proving that the Lock-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

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

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

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

- 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)

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

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

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

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

Queue Predicate, Invariant, and Proof of Hoare triples

In Coq!