

# 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**

- 1 Queue Specifications (HOCAP)
- 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-Free Michael-Scott Queue Satisfies the HOCAP-style Specification

## 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

$$\gamma \models_{\bullet} xS_v$$

Owned by queue

## Fragmental view

$$\gamma \models_{\circ} xS_v$$

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

$$\gamma \Vdash_{\bullet} xs_v$$

Owned by queue

## Fragmental view

$$\gamma \Vdash_{\circ} xs_v$$

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**:  $\text{AUTH}((\text{FRAC} \times \text{AG}(\text{List Val}))^?)$
- The **desirables** are captured by the following **lemmas**

## Lemmas on the Abstract State RA

$$\vdash \Vdash \exists \gamma. \gamma \Vdash_{\bullet} xs_v * \gamma \Vdash_{\circ} xs_v$$

(Abstract State Alloc)

$$\gamma \Vdash_{\bullet} xs'_v * \gamma \Vdash_{\circ} xs_v \vdash xs_v = xs'_v$$

(Abstract State Agree)

$$\gamma \Vdash_{\bullet} xs'_v * \gamma \Vdash_{\circ} xs_v \Rightarrow \gamma \Vdash_{\bullet} xs''_v * \gamma \Vdash_{\circ} xs_v$$

(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)

$\exists \text{isQueue} : \text{Val} \rightarrow \text{Qgnames} \rightarrow \text{Prop}.$

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

# 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)

$\exists \text{isQueue} : Val \rightarrow Qnames \rightarrow \text{Prop}.$

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

$\wedge \{ \text{True} \} \text{ initialize } () \{ v_q. \exists G. \text{isQueue}(v_q, G) * G. \gamma_{\text{Abst}} \mapsto_o [] \}$

# 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)

$\exists \text{isQueue} : \text{Val} \rightarrow \text{Qnames} \rightarrow \text{Prop.}$

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

$\wedge \{ \text{True} \} \text{ initialize } () \{ v_q. \exists G. \text{isQueue}(v_q, G) * G.\gamma_{\text{Abst}} \mapsto_o [] \}$

$\wedge \forall v_q, v, G, P, Q. \left( \forall xs_v. G.\gamma_{\text{Abst}} \mapsto_{\bullet} xs_v * P \Rightarrow_{\mathcal{E} \setminus \mathcal{N}.i \uparrow} \triangleright G.\gamma_{\text{Abst}} \mapsto_{\bullet} (v :: xs_v) * Q \right) \multimap$   
 $\{ \text{isQueue}(v_q, G) * P \} \text{ enqueue } v_q \ v \{ w.Q \}$



# 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)

$\exists \text{isQueue} : Val \rightarrow Qnames \rightarrow \text{Prop}.$

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

$\wedge \quad \{\text{True}\} \text{ initialize } () \{v_q. \exists G. \text{isQueue}(v_q, G) * G.\gamma_{\text{Abst}} \mapsto_o []\}$

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

$\wedge \quad \forall v_q, G, P, Q.$

$\left( \forall xs_v. G.\gamma_{\text{Abst}} \mapsto_{\bullet} xs_v * P \Rightarrow_{\mathcal{E} \setminus \mathcal{N}.i \uparrow} \triangleright \left( \begin{array}{l} (xs_v = [] * G.\gamma_{\text{Abst}} \mapsto_{\bullet} xs_v * Q(\text{None})) \\ \vee \left( \begin{array}{l} \exists v, xs'_v. xs_v = xs'_v ++ [v] * \\ G.\gamma_{\text{Abst}} \mapsto_{\bullet} xs'_v * Q(\text{Some } v) \end{array} \right) \end{array} \right) \right) \multimap$   
 $\{\text{isQueue}(v_q, G) * P\} \text{ dequeue } v_q \{w.Q(w)\}$

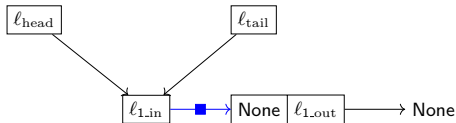
## The Two-Lock Michael-Scott Queue

## 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

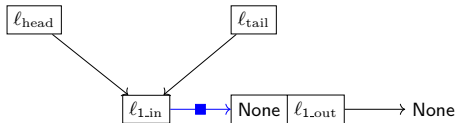
$$\text{in}(x) = \ell_{\text{in}}$$

$$\text{val}(x) = w$$

$$\text{out}(x) = \ell_{\text{out}}$$

`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

$$\text{in}(x) = \ell_{\text{in}}$$

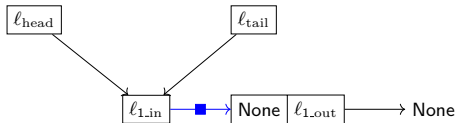
$$\text{val}(x) = w$$

$$\text{out}(x) = \ell_{\text{out}}$$

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

**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))
```

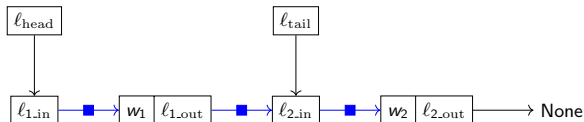


## Implementation: enqueue

- The **enqueue** function consists of the following **steps**

- 1 Create a new node,  $x_{\text{new}}$ , containing value to be enqueued
- 2 Acquire the tail lock
- 3 Add  $x_{\text{new}}$  to linked list
- 4 Swing tail pointer to  $x_{\text{new}}$
- 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)))
```

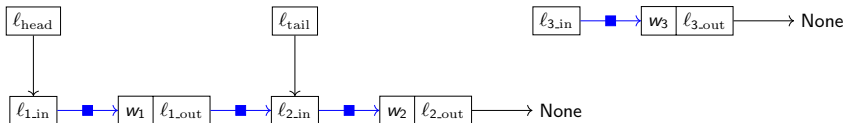


## Implementation: enqueue

- The **enqueue** function consists of the following **steps**

- 1 Create a new node,  $x_{\text{new}}$ , containing value to be enqueued
- 2 Acquire the tail lock
- 3 Add  $x_{\text{new}}$  to linked list
- 4 Swing tail pointer to  $x_{\text{new}}$
- 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)))
```



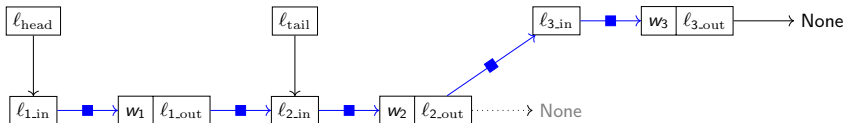
## Implementation: enqueue

- The **enqueue** function consists of the following **steps**

- 1 Create a new node,  $x_{\text{new}}$ , containing value to be enqueued
- 2 Acquire the tail lock
- 3 Add  $x_{\text{new}}$  to linked list
- 4 Swing tail pointer to  $x_{\text{new}}$
- 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)))) ← node;
  snd(fst(! Q)) ← node;
  release(snd(snd(! Q)))
```



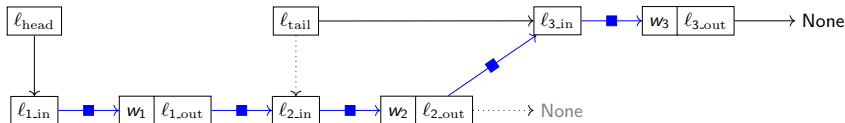


## Implementation: enqueue

- The **enqueue** function consists of the following **steps**

- 1 Create a new node,  $x_{\text{new}}$ , containing value to be enqueued
- 2 Acquire the tail lock
- 3 Add  $x_{\text{new}}$  to linked list
- 4 Swing tail pointer to  $x_{\text{new}}$
- 5 Release the tail lock

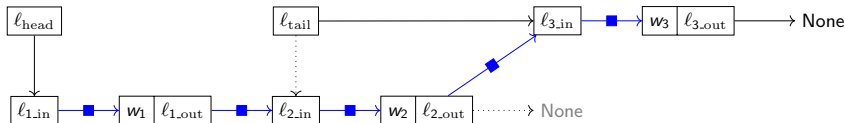
```
enqueue  $Q$  value  $\triangleq$   
let node = ref (Some value, ref (None)) in  
  acquire(snd(snd(! Q)));  
  snd(! (snd(fst(! Q))))  $\leftarrow$  node;  
  snd(fst(! Q))  $\leftarrow$  node;  
  release(snd(snd(! Q)))
```



## Implementation: enqueue

- The **enqueue** function consists of the following **steps**
  - 1 Create a new node,  $x_{\text{new}}$ , containing value to be enqueued
  - 2 Acquire the tail lock
  - 3 Add  $x_{\text{new}}$  to linked list
  - 4 Swing tail pointer to  $x_{\text{new}}$
  - 5 Release the tail lock
- Once a node is enqueued, its **position** in the linked list is **fixed**

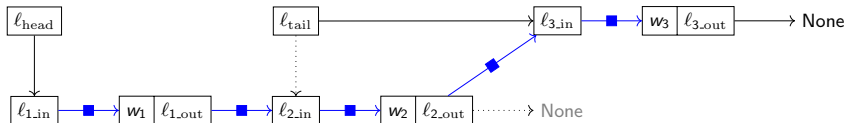
```
enqueue  $Q$  value  $\triangleq$   
let node = ref (Some value, ref (None)) in  
  acquire(snd(snd(! Q)));  
  snd(! (snd(fst(! Q))))  $\leftarrow$  node;  
  snd(fst(! Q))  $\leftarrow$  node;  
  release(snd(snd(! Q)))
```



## Implementation: enqueue

- The **enqueue** function consists of the following **steps**
  - 1 Create a new node,  $x_{\text{new}}$ , containing value to be enqueued
  - 2 Acquire the tail lock
  - 3 Add  $x_{\text{new}}$  to linked list
  - 4 Swing tail pointer to  $x_{\text{new}}$
  - 5 Release the tail lock
- Once a node is enqueued, its **position** in the linked list is **fixed**
- Adding and swinging **not atomic** → **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)))
```

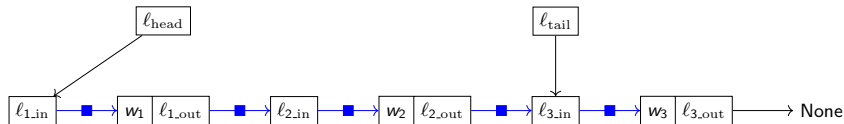




## Implementation: dequeue

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

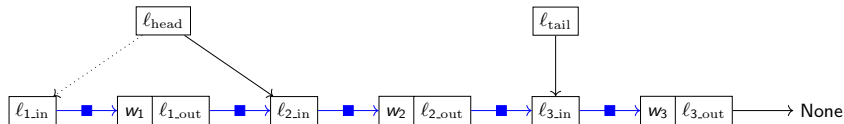
```
dequeue Q  $\triangleq$   
  acquire(fst(snd(! Q)));  
  let node = !(fst(fst(! Q))) in  
  let new_head = !(snd(! node)) in  
  if new_head = None then  
    release(fst(snd(! Q)));  
    None  
  else  
    let value = fst(! new_head) in  
    fst(fst(! Q))  $\leftarrow$  new_head;  
    release(fst(snd(! 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

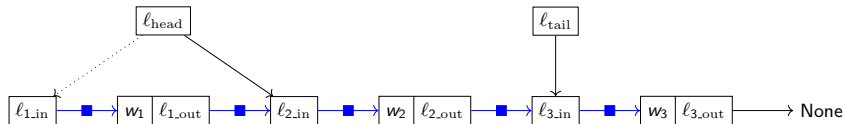
```
dequeue Q  $\triangleq$   
  acquire(fst(snd(! Q)));  
  let node = !(fst(fst(! Q))) in  
  let new_head = !(snd(! node)) in  
  if new_head = None then  
    release(fst(snd(! Q)));  
    None  
  else  
    let value = fst(! new_head) in  
    fst(fst(! Q))  $\leftarrow$  new_head;  
    release(fst(snd(! 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  $\triangleq$   
  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))  $\leftarrow$  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)

$$\text{isLL\_chain}([]) \triangleq \text{True}$$

$$\text{isLL\_chain}([x]) \triangleq \text{in}(x) \mapsto^{\square} (\text{val}(x), \text{out}(x))$$

$$\text{isLL\_chain}(x :: x' :: xs) \triangleq \text{in}(x) \mapsto^{\square} (\text{val}(x), \text{out}(x)) * \text{out}(x') \mapsto^{\square} \text{in}(x') * \text{isLL\_chain}(x' :: xs)$$

---

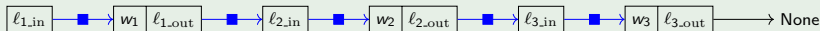

$$\text{isLL}([]) \triangleq \text{True}$$

$$\text{isLL}(x :: xs) \triangleq \text{out}(x) \mapsto \text{None} * \text{isLL\_chain}(x :: xs)$$

## Example

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

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



# Invariant

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

$$I_{\text{TLH}}(\ell_{\text{head}}, \ell_{\text{tail}}, G) \triangleq \exists x s_v. G. \gamma_{\text{Abst}} \Rightarrow_{\bullet} x s_v \ast \quad (\text{abstract state})$$

# Invariant

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

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

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

# 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,  $xs$
  - Asserts relation between abstract state and concrete state

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

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

# 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,  $xs$
  - 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)

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

## 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  $x_{\text{new}}$  and swinging  $\ell_{\text{tail}}$
- **Tokens** used to reason about which state queue is in

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

...

$\ell_{\text{head}} \mapsto \text{in}(x_{\text{head}}) * \ell_{\text{tail}} \mapsto \text{in}(x_{\text{tail}}) * \text{isLast}(x_{\text{tail}}, xs) *$  (Static)

$\text{TokNE } G * \text{TokND } G * \text{TokUpdated } G$

✓  $\ell_{\text{head}} \mapsto \text{in}(x_{\text{head}}) * \ell_{\text{tail}} \mapsto \frac{1}{2} \text{in}(x_{\text{tail}}) *$  (Enqueue)

$(\text{isLast}(x_{\text{tail}}, xs) * \text{TokBefore } G \vee \text{isSndLast}(x_{\text{tail}}, xs) * \text{TokAfter } G) *$

$\text{TokE } G * \text{TokND } G$

✓  $\ell_{\text{head}} \mapsto \frac{1}{2} \text{in}(x_{\text{head}}) * \ell_{\text{tail}} \mapsto \text{in}(x_{\text{tail}}) * \text{isLast}(x_{\text{tail}}, xs) *$  (Dequeue)

$\text{TokNE } G * \text{TokD } G * \text{TokUpdated } G$

✓  $\ell_{\text{head}} \mapsto \frac{1}{2} \text{in}(x_{\text{head}}) * \ell_{\text{tail}} \mapsto \frac{1}{2} \text{in}(x_{\text{tail}}) *$  (Both)

$(\text{isLast}(x_{\text{tail}}, xs) * \text{TokBefore } G \vee \text{isSndLast}(x_{\text{tail}}, xs) * \text{TokAfter } G) *$

$\text{TokE } G * \text{TokD } G$

## Queue Predicate

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

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

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



## Queue Predicate

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

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

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

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

# Queue Predicate

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

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

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

- 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

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

# Proof Sketch of the Hoare Triple for enqueue

$$\forall v_q, v, G, P, Q. \left( \forall x_{s_v}. G.\gamma_{\text{Abst}} \models \bullet x_{s_v} * P \Rightarrow_{\mathcal{E} \setminus \mathcal{N}.i \uparrow} \triangleright G.\gamma_{\text{Abst}} \models \bullet (v :: x_{s_v}) * Q \right) \multimap \{ \text{isQueue}(v_q, G) * P \} \text{enqueue } v_q \ v \{ w.Q \}$$

(Proof)

---

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

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

# Proof Sketch of the Hoare Triple for enqueue

$\forall v_q, v, G, P, Q. (\forall x s_v. G.\gamma_{\text{Abst}} \mapsto \bullet x s_v * P \Rightarrow_{\mathcal{E} \setminus \mathcal{N}.i \uparrow} \triangleright G.\gamma_{\text{Abst}} \mapsto \bullet (v :: x s_v) * Q) \multimap$   
 $\{\text{isQueue}(v_q, G) * P\} \text{enqueue } v_q \ v \ \{w.Q\}$

(Proof)

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

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

$\{P\}$

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

`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

$\forall v_q, v, G, P, Q. (\forall x_{sv}. G.\gamma_{\text{Abst}} \models \bullet. x_{sv} * P \Rightarrow_{\mathcal{E} \setminus \mathcal{N}.i \uparrow} \triangleright G.\gamma_{\text{Abst}} \models \bullet. (v :: x_{sv}) * Q) \multimap$   
 $\{\text{isQueue}(v_q, G) * P\} \text{enqueue } v_q \ v \ \{w.Q\}$

(Proof)

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

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

$\{P\}$

let  $node = \text{ref}(\text{Some } v, \text{ref}(\text{None}))$  in (create node  $x_{\text{new}}$ )

$\{P * \text{out}(x_{\text{new}}) \mapsto \text{None}\}$

acquire( $\text{snd}(\text{snd}(!v_q))$ );

$e_t = !(\text{snd}(\text{fst}(!v_q)))$

$\text{snd}(!e_t) \leftarrow node$ ;

$\text{snd}(\text{fst}(!v_q)) \leftarrow node$ ;

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

$\{Q\}$

# Proof Sketch of the Hoare Triple for enqueue

$\forall v_q, v, G, P, Q. (\forall x_{sv}. G.\gamma_{\text{Abst}} \mapsto \bullet x_{sv} * P \Rightarrow_{\mathcal{E} \setminus \mathcal{N}.i \uparrow} \triangleright G.\gamma_{\text{Abst}} \mapsto \bullet (v :: x_{sv}) * Q) \multimap$   
 $\{\text{isQueue}(v_q, G) * P\} \text{ enqueue } v_q \ v \ \{w.Q\}$

(Proof)

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

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

$\{P\}$

let  $node = \text{ref}(\text{Some } v, \text{ref}(\text{None}))$  in (create node  $x_{\text{new}}$ )

$\{P * \text{out}(x_{\text{new}}) \mapsto \text{None}\}$

acquire( $\text{snd}(\text{snd}(!v_q))$ ); (acquire tail lock)

$\{P * \text{out}(x_{\text{new}}) \mapsto \text{None} * \text{TokE } G\}$

$e_t = !(\text{snd}(\text{fst}(!v_q)))$

$\text{snd}(!e_t) \leftarrow node;$

$\text{snd}(\text{fst}(!v_q)) \leftarrow node;$

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

$\{Q\}$

# Proof Sketch of the Hoare Triple for enqueue

$$\forall v_q, v, G, P, Q. (\forall x_{sv}. G. \gamma_{\text{Abst}} \mapsto \bullet. x_{sv} * P \Rightarrow_{\mathcal{E} \setminus \mathcal{N}.i \uparrow} \triangleright G. \gamma_{\text{Abst}} \mapsto \bullet. (v :: x_{sv}) * Q) \multimap \{ \text{isQueue}(v_q, G) * P \} \text{ enqueue } v_q \ v \{ w. Q \}$$

(Proof)

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

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

```
{P}
  let node = ref(Some v, ref(None)) in (create node xnew)
{P * out(xnew) ↦ None}
  acquire(snd(snd(! vq))); (acquire tail lock)
{P * out(xnew) ↦ None * TokE G}
  et = !(snd(fst(! vq))) (find current tail, xtail. ITLH: Static/Dequeue → Enqueue/Both (before))
{P * out(xnew) ↦ None *  $\ell_{\text{tail}} \mapsto \frac{1}{2}$  in(xtail) * TokNE G * TokAfter G}
  snd(! (et)) ← node;

  snd(fst(! vq)) ← node;

  release(snd(snd(! vq)))
{Q}
```



# Proof Sketch of the Hoare Triple for enqueue

$$\forall v_q, v, G, P, Q. \quad (\forall x_{s_v}. G. \gamma_{\text{Abst}} \mapsto \bullet. x_{s_v} * P \Rightarrow_{\mathcal{E} \setminus \mathcal{N}. i \uparrow} \triangleright G. \gamma_{\text{Abst}} \mapsto \bullet. (v :: x_{s_v}) * Q) \multimap \{ \text{isQueue}(v_q, G) * P \} \text{ enqueue } v_q \ v \{ w. Q \}$$

(Proof)

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

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

```
{P}
  let node = ref(Some v, ref(None)) in   (create node xnew)
{P * out(xnew) ↦ None}
  acquire(snd(snd(! vq)));   (acquire tail lock)
{P * out(xnew) ↦ None * TokE G}
  et = !(snd(fst(! vq)))   (find current tail, xtail. ITLH: Static/Dequeue → Enqueue/Both (before))
{P * out(xnew) ↦ None * ℓtail ↦ 1/2 in(xtail) * TokNE G * TokAfter G}
  snd(! (et)) ← node;   (make xtail point to xnew. ITLH: Enqueue/Both (before) → Enqueue/Both (after))
{Q * ℓtail ↦ 1/2 in(xtail) * TokNE G * TokBefore G}
  snd(fst(! vq)) ← node;

  release(snd(snd(! vq)))
{Q}
```

# Proof Sketch of the Hoare Triple for enqueue

$$\forall v_q, v, G, P, Q. (\forall x_{sv}. G. \gamma_{\text{Abst}} \mapsto \bullet. x_{sv} * P \Rightarrow_{\mathcal{E} \setminus \mathcal{N}.i \uparrow} \triangleright G. \gamma_{\text{Abst}} \mapsto \bullet. (v :: x_{sv}) * Q) \multimap \{ \text{isQueue}(v_q, G) * P \} \text{ enqueue } v_q \ v \{ w. Q \}$$

(Proof)

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

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

```
{P}
  let node = ref(Some v, ref(None)) in (create node xnew)
{P * out(xnew) ↦ None}
  acquire(snd(snd(! vq))); (acquire tail lock)
{P * out(xnew) ↦ None * TokE G}
  et = !(snd(fst(! vq))) (find current tail, xtail. ITLH: Static/Dequeue → Enqueue/Both (before))
{P * out(xnew) ↦ None * ℓtail ↦ 1/2 in(xtail) * TokNE G * TokAfter G}
  snd(! (et)) ← node; (make xtail point to xnew. ITLH: Enqueue/Both (before) → Enqueue/Both (after))
{Q * ℓtail ↦ 1/2 in(xtail) * TokNE G * TokBefore G}
  snd(fst(! vq)) ← node; (swing tail pointer to xnew. ITLH: Enqueue/Both (after) → Static/Dequeue)
{Q * TokE G}
  release(snd(snd(! vq)))
{Q}
```

# Proof Sketch of the Hoare Triple for enqueue

$$\forall v_q, v, G, P, Q. (\forall x_{sv}. G. \gamma_{\text{Abst}} \mapsto \bullet. x_{sv} * P \Rightarrow_{\mathcal{E} \setminus \mathcal{N}.i \uparrow} \triangleright G. \gamma_{\text{Abst}} \mapsto \bullet. (v :: x_{sv}) * Q) \multimap \{ \text{isQueue}(v_q, G) * P \} \text{ enqueue } v_q \ v \{ w. Q \}$$

(Proof)

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

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

```
{P}
  let node = ref(Some v, ref(None)) in (create node xnew)
{P * out(xnew) ↦ None}
  acquire(snd(snd(! vq))); (acquire tail lock)
{P * out(xnew) ↦ None * TokE G}
  et = !(snd(fst(! vq))) (find current tail, xtail. ITLH: Static/Dequeue → Enqueue/Both (before))
{P * out(xnew) ↦ None * ℓtail ↦ 1/2 in(xtail) * TokNE G * TokAfter G}
  snd(! (et)) ← node; (make xtail point to xnew. ITLH: Enqueue/Both (before) → Enqueue/Both (after))
{Q * ℓtail ↦ 1/2 in(xtail) * TokNE G * TokBefore G}
  snd(fst(! vq)) ← node; (swing tail pointer to xnew. ITLH: Enqueue/Both (after) → Static/Dequeue)
{Q * TokE G}
  release(snd(snd(! vq))) (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  $\triangleq$   
  let node = ref (None, ref (None)) in  
  ref (ref (node), ref (node))  
  
enqueue Q value  $\triangleq$   
  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  $\triangleq$   
  (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  $\triangleq$   
  let node = ref (None, ref (None)) in  
  ref (ref (node), ref (node))  
  
enqueue Q value  $\triangleq$   
  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  $\triangleq$   
  (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 ()  
    )()
```

## 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** (à 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** (à 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** (à 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

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

# 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** (à 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

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

- 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.  $\gamma_{\text{Head}}$ ,  $\gamma_{\text{Tail}}$ , and  $\gamma_{\text{Last}}$ 
  - Implemented using the resource algebra  $\text{AUTH}(\mathcal{P}(\text{Node}))$



## 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.  $\gamma_{\text{Head}}$ ,  $\gamma_{\text{Tail}}$ , and  $\gamma_{\text{Last}}$ 
  - Implemented using the resource algebra  $\text{AUTH}(\mathcal{P}(\text{Node}))$
- **Defined** in two parts: **Abstract Points-to** ( $\gamma \multimap x$ ) and **Abstract Reach** ( $x \dashrightarrow \gamma$ )
- For instance,  $\gamma_{\text{Tail}} \multimap x_n$  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.  $\gamma_{\text{Head}}$ ,  $\gamma_{\text{Tail}}$ , and  $\gamma_{\text{Last}}$ 
  - Implemented using the resource algebra  $\text{AUTH}(\mathcal{P}(\text{Node}))$
- **Defined** in two parts: **Abstract Points-to** ( $\gamma \multimap x$ ) and **Abstract Reach** ( $x \dashrightarrow \gamma$ )
- For instance,  $\gamma_{\text{Tail}} \multimap x_n$  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)

$$x \leadsto x \Rightarrow \exists \gamma. \gamma \multimap x \quad (\text{Abs Reach Alloc})$$

$$x_n \dashrightarrow \gamma_m * \gamma_m \multimap x_m \multimap x_n \leadsto x_m \quad (\text{Abs Reach Concr})$$

$$x_n \leadsto x_m * \gamma_m \multimap x_m \Rightarrow x_n \dashrightarrow \gamma_m \quad (\text{Abs Reach Abs})$$

$$\gamma_m \multimap x_m * x_m \leadsto x_o \Rightarrow \gamma_m \multimap x_o \quad (\text{Abs Reach Advance})$$

# In Coq!

