

Master's Thesis Exam

Verification of the Blocking and Non-Blocking Michael-Scott Queue Algorithms

Mathias Pedersen, 201808137

Advisor: Amin Timany

Aarhus University

June 2024



AARHUS
UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

Overview of the Project and Contributions

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

- 1 Queue Specifications
- 2 The Two-Lock Michael-Scott Queue
- 3 Proving that the Two-Lock Michael-Scott Queue Satisfies the HOCAP-style Specification
- 4 The Lock-Free Michael-Scott Queue
- 5 Proving that the Lock-and-CC-Free Michael-Scott Queue Satisfies the HOCAP-style Specification

Queue Specifications

Specifications for Queues

Assumptions on Queues

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

Nature of Specifications

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

Sequential Specification

Definition (Sequential Specification)

$\exists \text{isQueue}_S : \text{Val} \rightarrow \text{List Val} \rightarrow \text{SeqQgnames} \rightarrow \text{Prop.}$

$\{\text{True}\} \text{ initialize } () \{v_q. \exists G. \text{isQueue}_S(v_q, [], G)\}$

$\wedge \quad \forall v_q, v, xs_v, G. \{\text{isQueue}_S(v_q, xs_v, G)\} \text{ enqueue } v_q \ v \{w. \text{isQueue}_S(v_q, (v :: xs_v), G)\}$

$\wedge \quad \forall v_q, xs_v, G. \{\text{isQueue}_S(v_q, xs_v, G)\}$

$\text{ dequeue } v_q$

$\left\{ w. \begin{array}{l} (xs_v = [] * w = \text{None} * \text{isQueue}_S(v_q, xs_v, G)) \vee \\ (\exists v, xs'_v. xs_v = xs'_v ++ [v] * w = \text{Some } v * \text{isQueue}_S(v_q, xs'_v, G)) \end{array} \right\}$

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

Concurrent Specification

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

Definition (Concurrent Specification)

$\exists \text{isQueue}_C : (Val \rightarrow Prop) \rightarrow Val \rightarrow ConcQnames \rightarrow Prop.$

$\forall \Psi : Val \rightarrow Prop.$

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

$\wedge \{True\} \text{ initialize } () \{v_q. \exists G. \text{isQueue}_C(\Psi, v_q, G)\}$

$\wedge \forall v_q, v, G. \{\text{isQueue}_C(\Psi, v_q, G) * \Psi(v)\} \text{ enqueue } v_q \ v \{w.True\}$

$\wedge \forall v_q, G. \{\text{isQueue}_C(\Psi, v_q, G)\} \text{ dequeue } v_q \{w. w = None \vee (\exists v. w = \text{Some } v * \Psi(v))\}$

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

Definition (HOCAP Specification)

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

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

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

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

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

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

Queue Client - A PoC Client

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

```
unwrap w  $\triangleq$  match w with None  $\Rightarrow$  () () | Some v  $\Rightarrow$  v end
```

```
enqdeq vq c  $\triangleq$  enqueue vq c; unwrap(dequeue vq)
```

```
queueAdd a b  $\triangleq$   
  let vq = initialize () in  
  let p = (enqdeq vq a) || (enqdeq vq b) in  
  fst p + snd p
```

Queue Client - A PoC Client (continued)

Lemma (QueueAdd Specification)

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

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

Definition (Invariant for QueueAdd)

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

The Two-Lock Michael-Scott Queue

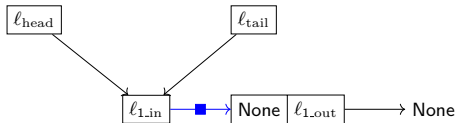
Implementation: initialize

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

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

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

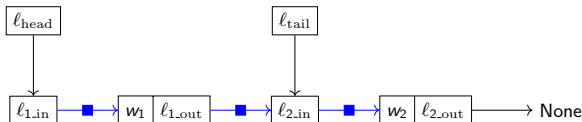
```
initialize  $\triangleq$   
  let node = ref (None, ref (None)) in  
  let H_lock = newLock() in  
  let T_lock = newLock() in  
  ref ((ref (node), ref (node)), (H_lock, T_lock))
```



Implementation: enqueue

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

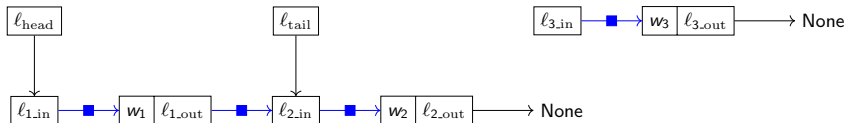
```
enqueue  $Q$  value  $\triangleq$   
  let node = ref (Some value, ref (None)) in  
  acquire(snd(snd(! Q)));  
  snd(! (snd(fst(! Q))))  $\leftarrow$  node;  
  snd(fst(! Q))  $\leftarrow$  node;  
  release(snd(snd(! Q)))
```



Implementation: 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

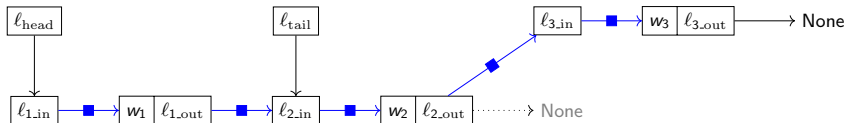
```
enqueue  $Q$  value  $\triangleq$   
  let node = ref (Some value, ref (None)) in  
  acquire(snd(snd(! Q)));  
  snd(! (snd(fst(! Q))))  $\leftarrow$  node;  
  snd(fst(! Q))  $\leftarrow$  node;  
  release(snd(snd(! Q)))
```



Implementation: 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

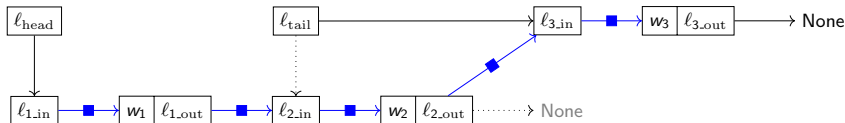
```
enqueue  $Q$  value  $\triangleq$   
let node = ref (Some value, ref (None)) in  
  acquire(snd(snd(! Q)));  
  snd(! (snd(fst(! Q))))  $\leftarrow$  node;  
  snd(fst(! Q))  $\leftarrow$  node;  
  release(snd(snd(! Q)))
```



Implementation: 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

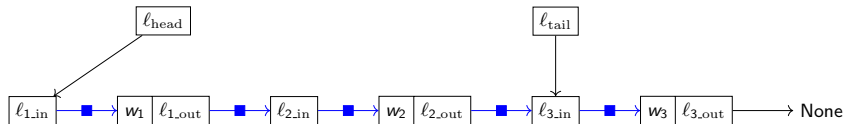
```
enqueue  $Q$  value  $\triangleq$   
let node = ref (Some value, ref (None)) in  
  acquire(snd(snd(! Q)));  
  snd(! (snd(fst(! Q))))  $\leftarrow$  node;  
  snd(fst(! Q))  $\leftarrow$  node;  
  release(snd(snd(! Q)))
```



Implementation: dequeue

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

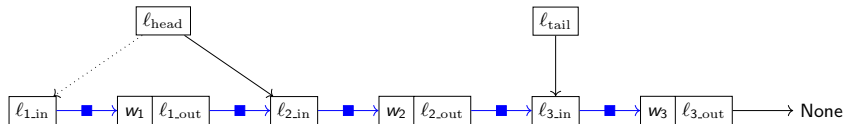
```
dequeue  $Q \triangleq$   
  acquire(fst(snd(! Q)));  
  let node = !(fst(fst(! Q))) in  
  let new_head = !(snd(! node)) in  
  if new_head = None then  
    release(fst(snd(! Q)));  
    None  
  else  
    let value = fst(! new_head) in  
    fst(fst(! Q))  $\leftarrow$  new_head;  
    release(fst(snd(!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



Implementation: dequeue

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

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

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

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

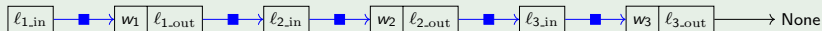
$$\text{isLL}([]) \triangleq \text{True}$$

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

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)
- The queue relies on non-persistent resources (e.g. $\ell_{\text{head}} \mapsto \ell_{\text{in}}$)
- Solution: identify an *invariant* (persistent), describing the resources

Invariant

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

$$\begin{aligned} I_{\text{TLH}}(\ell_{\text{head}}, \ell_{\text{tail}}, G) &\triangleq \exists x_{\text{sv}}. G.\gamma_{\text{Abst}} \mapsto_{\bullet} x_{\text{sv}} * \\ &\quad \exists x_{\text{s}}, x_{\text{s}_{\text{queue}}}, x_{\text{s}_{\text{old}}}, x_{\text{head}}, x_{\text{tail}}. \\ &\quad x_{\text{s}} = x_{\text{s}_{\text{queue}}} ++ [x_{\text{head}}] ++ x_{\text{s}_{\text{old}}} * \\ &\quad \text{isLL}(x_{\text{s}}) * \\ &\quad \text{projVal}(x_{\text{s}_{\text{queue}}}) = \text{wrapSome}(x_{\text{sv}}) * \\ &\quad \dots \end{aligned}$$

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_{new} and swinging ℓ_{tail}
- Tokens used to reason about which state queue is in

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

...

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

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

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

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

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

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

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

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

Proof Sketch of the Hoare Triple for enqueue

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

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 $\text{isQueue}(v_q, Qgnames)$:

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

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 $\text{isQueue}(v_q, Qgnames)$:

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

```
{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

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

(Proof)

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

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

$\{P\}$

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

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

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

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

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

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

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

$\{Q\}$

Proof Sketch of the Hoare Triple for enqueue

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

(Proof)

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

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

```
{P}
let node = 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)))

  snd(! (et)) ← node;

  snd(fst(! vq)) ← node;

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

Proof Sketch of the Hoare Triple for enqueue

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

(Proof)

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

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

```
{P}
  let node = ref(Some v, ref(None)) in (create node 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;

  snd(fst(! vq)) ← node;

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

Proof Sketch of the Hoare Triple for enqueue

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

(Proof)

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

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

```
{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

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

(Proof)

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

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

```
{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

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

(Proof)

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

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

```
{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}
```

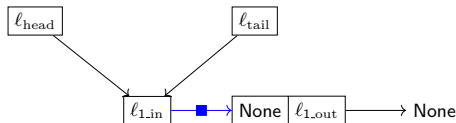
The Lock-Free Michael-Scott Queue

Implementation: initialize

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

initialize \triangleq

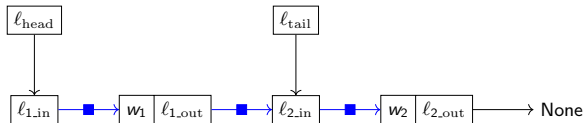
```
let node = ref (None, ref (None)) in  
ref (ref (node), ref (node))
```



Implementation: enqueue

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

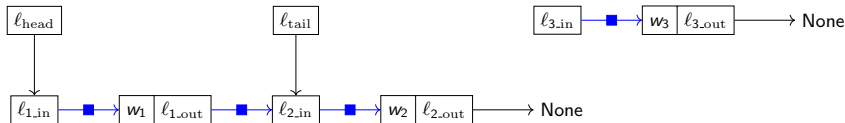
```
enqueue  $Q$  value  $\triangleq$   
let node = ref (Some value, ref (None)) in  
(rec loop_ =  
  let tail = !(snd(! Q)) in  
  let next = !(snd(! tail)) in  
  if tail = !(snd(! Q)) then  
    if next = None then  
      if CAS (snd(! tail)) next node then  
        CAS (snd(! Q)) tail node  
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



Implementation: enqueue

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

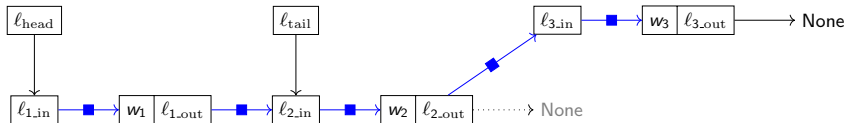
```
enqueue  $Q$  value  $\triangleq$   
let node = ref (Some value, ref (None)) in  
(rec loop_ =  
  let tail = !(snd(! Q)) in  
  let next = !(snd(! tail)) in  
  if tail = !(snd(! Q)) then  
    if next = None then  
      if CAS (snd(! tail)) next node then  
        CAS (snd(! Q)) tail node  
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



Implementation: enqueue

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

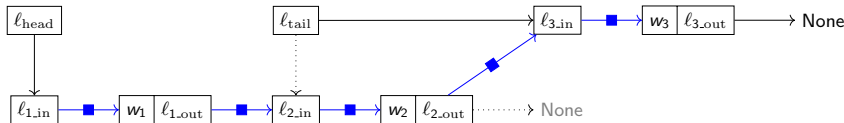
```
enqueue  $Q$  value  $\triangleq$   
let node = ref (Some value, ref (None)) in  
(rec loop_ =  
  let tail = !(snd(! Q)) in  
  let next = !(snd(! tail)) in  
  if tail = !(snd(! Q)) then  
    if next = None then  
      if CAS (snd(! tail)) next node then  
        CAS (snd(! Q)) tail node  
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



Implementation: enqueue

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

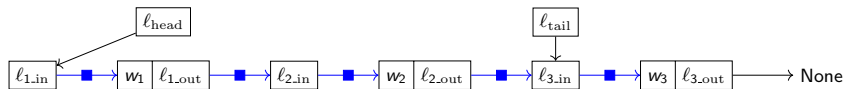
```
enqueue  $Q$  value  $\triangleq$   
let node = ref (Some value, ref (None)) in  
(rec loop_ =  
  let tail = !(snd(! Q)) in  
  let next = !(snd(! tail)) in  
  if tail = !(snd(! Q)) then  
    if next = None then  
      if CAS (snd(! tail)) next node then  
        CAS (snd(! Q)) tail node  
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



Implementation: dequeue

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

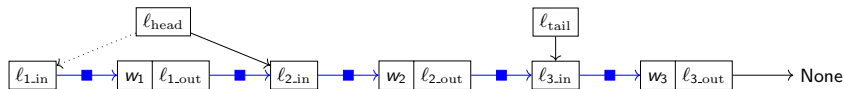
```
dequeue Q  $\triangleq$   
(rec loop_ =  
  let head = !(fst(! Q)) in  
  let tail = !(snd(! Q)) in  
  let p = newproph in  
  let next = !(snd(! head)) in  
  if head = Resolve(! (fst(! Q)), p, ()) then  
    if head = tail then  
      if next = None then  
        None  
      else  
       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



Implementation: dequeue

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

```
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_ ()  
  else loop_ ()  
)()
```



Prophecies

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

```
...
let p = newproph in
let next = !(snd(! head)) in
if head = Resolve(!(fst(! Q)), p, ()) then
  if head = tail then
    if next = None then
      None
  ...
else loop ()
...
```

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

- Reason for consistency checks: ABA problem in original implementation
- HeapLang is garbage collected language, so we can remove consistency checks
- Can also remove prophecy in dequeue
 - When we read *next*, we know immediately whether dequeue will conclude empty queue
 - both *head* and *tail* are already fixed
- Correctness: both versions shown to satisfy HOCAP-style specification...

```
initialize  $\triangleq$   
  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 ()  
  ) ()
```

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

Reachability

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

Concrete Reachability

- Concrete reachability essentially captures a section of the linked list (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
- Tracked using ghost names, e.g. γ_{Head} , γ_{Tail} , and γ_{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!

