Master's Thesis Exam
Verification of the Blocking and Non-Blocking Michael-Scott Queue Algorithms

Mathias Pedersen, 201808137
Advisor: Amin Timany

Aarhus University

June 2024

AARHUS UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Overview of the Project and Contributions

▶create slide◀

Outline

Queue Specifications

Specifications for Queues

Assumptions on Queues

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

Nature of Specifications

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

Sequential Specification

Definition (Sequential Specification)

$$
\begin{array}{ll}
\multicolumn{2}{l}{\exists\, \mathrm{isQueue}_S : \mathit{Val} \to \mathit{List\ Val} \to \mathit{SeqQgnames} \to \mathrm{Prop}.} \\
& \{\mathrm{True}\}\ \mathrm{initialize}\ ()\ \{v_q. \exists G.\ \mathrm{isQueue}_S(v_q, [], G)\} \\
\wedge & \forall v_q, v, xs_v, G.\ \{\mathrm{isQueue}_S(v_q, xs_v, G)\}\ \mathrm{enqueue}\ \ v_q\ \ v\ \{w.\ \mathrm{isQueue}_S(v_q, (v :: xs_v), G)\} \\
\wedge & \forall v_q, xs_v, G.\ \{\mathrm{isQueue}_S(v_q, xs_v, G)\} \\
& \qquad\qquad \mathrm{dequeue}\ \ v_q \\
& \qquad \left\{ w.\ \begin{array}{l} (xs_v = [] * w = \mathrm{None} * \mathrm{isQueue}_S(v_q, xs_v, G)) \vee \\ (\exists v, xs_v'.\ xs_v = xs_v' +\!\!+ [v] * w = \mathrm{Some}\ v *\ \mathrm{isQueue}_S(v_q, xs_v', G)) \end{array} \right\}
\end{array}
$$

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

Concurrent Specification

Definition (Concurrent Specification)

$$\exists\, \text{isQueue}_C : (\mathit{Val} \to \text{Prop}) \to \mathit{Val} \to \mathit{ConcQgnames} \to \text{Prop}.$$

$$\forall \Psi : \mathit{Val} \to \text{Prop}.$$

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

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

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

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

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

Lemmas on the Abstract State RA

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

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

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

HOCAP-style Specification

Definition (HOCAP Specification)

$$
\begin{array}{ll}
\multicolumn{2}{l}{\exists\, \text{isQueue} : \mathit{Val} \to \mathit{Qgnames} \to \text{Prop}.} \\
 & \forall v_q, G.\ \text{isQueue}(v_q, G) \implies \Box\, \text{isQueue}(v_q, G) \\
\wedge & \{\text{True}\}\ \text{initialize}\ ()\ \{v_q . \exists G.\ \text{isQueue}(v_q, G) * G.\gamma_{\mathrm{Abst}} \Mapsto_\circ []\} \\
\wedge & \forall v_q, v, G, P, Q.\ \ (\forall xs_v.\ G.\gamma_{\mathrm{Abst}} \Mapsto_\bullet xs_v * P \Rrightarrow_{\mathcal{E}\setminus\mathcal{N}.i\uparrow} \triangleright G.\gamma_{\mathrm{Abst}} \Mapsto_\bullet (v :: xs_v) * Q) \twoheadrightarrow \\
 & \qquad \{\text{isQueue}(v_q, G) * P\}\ \text{enqueue}\ \ v_q\ \ v\ \{w.Q\} \\
\wedge & \forall v_q, G, P, Q. \\
 & \left(\forall xs_v.\ G.\gamma_{\mathrm{Abst}} \Mapsto_\bullet xs_v * P \Rrightarrow_{\mathcal{E}\setminus\mathcal{N}.i\uparrow} \triangleright \left(\begin{array}{l} (xs_v = [] * G.\gamma_{\mathrm{Abst}} \Mapsto_\bullet xs_v * Q(\text{None})) \\ \vee \left(\begin{array}{l} \exists v, xs_v'.\ xs_v = xs_v' \mathbin{+\!+} [v] * \\ G.\gamma_{\mathrm{Abst}} \Mapsto_\bullet xs_v' * Q(\text{Some}\ v) \end{array}\right) \end{array}\right)\right. \\
 & \{\text{isQueue}(v_q, G) * P\}\ \text{dequeue}\ \ v_q\ \{w.Q(w)\}
\end{array}
$$

Queue Client - A PoC Client

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

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

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

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

Queue Client - A PoC Client (continued)

Lemma (QueueAdd Specification)

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

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

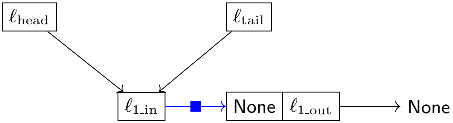
Definition (Invariant for QueueAdd)

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

The Two-Lock Michael-Scott Queue

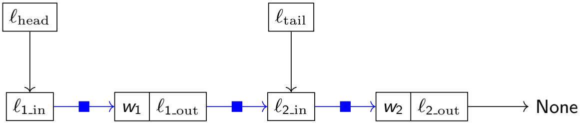
Implementation: initialize

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



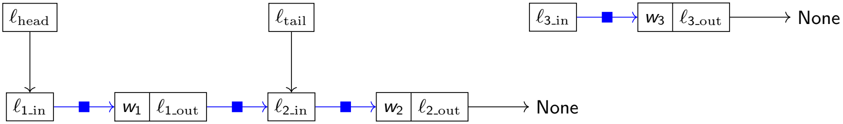
Implementation: enqueue

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



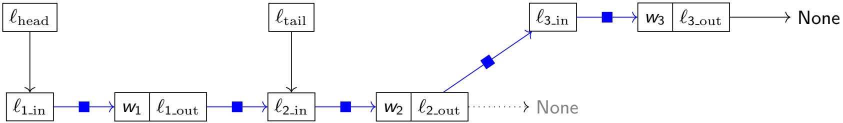
Implementation: enqueue

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



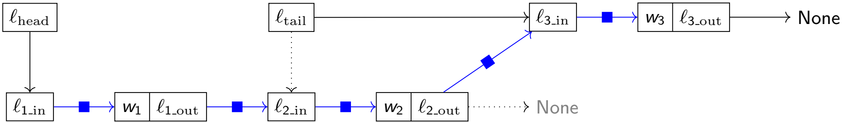
Implementation: enqueue

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



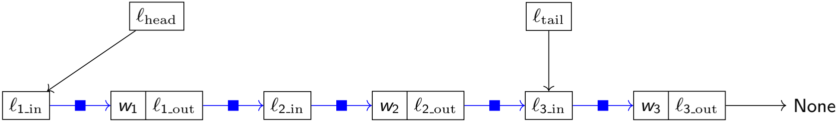
Implementation: enqueue

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

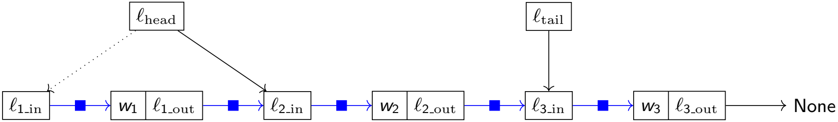


Implementation: dequeue

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



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



▶format and simplify◀

1. The tail node is always either the last or second last node in the linked list.
2. All but the last pointer in the linked list (the pointer to None) never change.
3. Nodes in the linked list are never deleted. Hence, the linked list only ever grows.
4. The tail can lag one node behind the head.
5. At any given time, the queue is in one of four states:
    1. No threads are interacting with the queue (**Static**).
    2. A thread is enqueueing (**Enqueue**).
    3. A thread is dequeueing (**Dequeue**).
    4. A thread is enqueueing and a thread is dequeueing (**Both**).

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

Invariant

►create slide◄

The isLL Predicate

►**create slide**◄

Queue Predicate

►**create slide**◄

Proof of Initialise

▶**create slide**◀

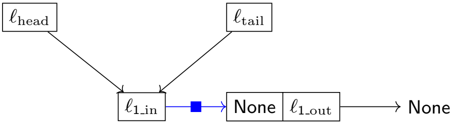
Proof of ▶enqueue **xor** dequeue◀

▶create slide◀

The Lock-Free Michael-Scott Queue

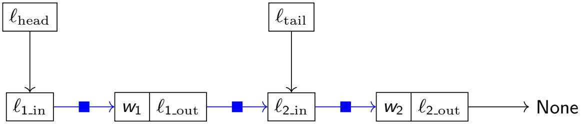
Implementation: initialize

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



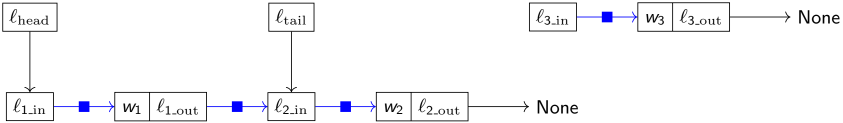
Implementation: enqueue

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



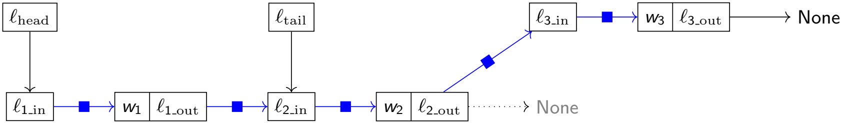
Implementation: enqueue

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



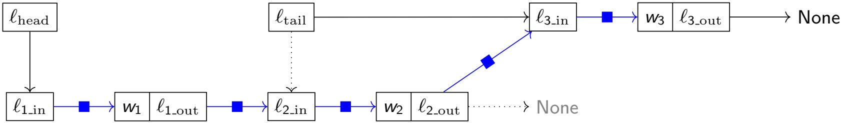
Implementation: enqueue

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



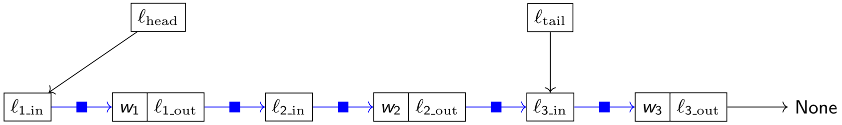
Implementation: enqueue

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



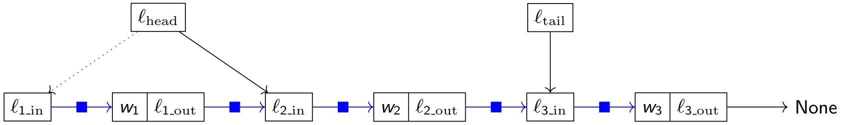
Implementation: dequeue

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



Implementation: dequeue

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



Prophecies

►create slide◄

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

▶format◀

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

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

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

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

Reachability

►**create slide**◄

Invariant

►create slide◄

Queue Predicate

►**create slide**◄

Proof of initialize and ►enqueue **xor** dequeue◄

In Coq!