

# The Best Queue Specifications You Will Ever See today probably

Mathias Pedersen

Aarhus University

November 2025



AARHUS  
UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

# Context

- Based on my Master's Thesis
- Goal of project was to prove **safety** of two concurrent queues
- Success! – but not too interesting

# Context

- Based on my Master's Thesis
- Goal of project was to prove safety of two concurrent queues
- Success! – but not too interesting
- Today: Queue Specifications

# Context

- Based on my Master's Thesis
- Goal of project was to prove safety of two concurrent queues
- Success! – but not too interesting
- Today: Queue Specifications
- In particular, three different specifications
  - Sequential specification
  - Concurrent specification
    - Doesn't track queue contents
  - HOCAP-style specification
    - Tracks queue contents with added complexity

# Context

- Based on my Master's Thesis
- Goal of project was to prove safety of two concurrent queues
- Success! – but not too interesting
- Today: Queue Specifications
- In particular, three different specifications
  - Sequential specification
  - Concurrent specification
    - Doesn't track queue contents
  - HOCAP-style specification
    - Tracks queue contents with added complexity
- Uses HeapLang, but should be mostly language-agnostic

# Context

- Based on my Master's Thesis
- **Goal** of project was to prove **safety** of two concurrent queues
- Success! – but not too interesting
- Today: Queue Specifications
- In particular, three different **specifications**
  - **Sequential** specification
  - **Concurrent** specification
    - **Doesn't track** queue contents
  - **HOCAP-style** specification
    - **Tracks** queue contents with added **complexity**
- Uses **HeapLang**, but should be mostly language-agnostic
- Project was advised by Amin

# Specifications for Queues

## Informal Queue Specification

- 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

## Informal Queue Specification

- 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**

# Sequential Specification

## Definition (Sequential Specification)

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

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

# Sequential Specification

## Definition (Sequential Specification)

$\exists \text{isQueue}_S : \text{Val} \rightarrow \text{List Val} \rightarrow \text{SeqQnames} \rightarrow \text{Prop}.$   
 $\{\text{True}\} \text{ initialize } () \{v_q. \exists G. \text{isQueue}_S(v_q, [], G)\}$

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

# Sequential Specification

## Definition (Sequential Specification)

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

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

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

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

# Sequential Specification

## Definition (Sequential Specification)

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

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

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

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

$\text{ dequeue } v_q$

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

- The proposition  $\text{isQueue}_S(v_q, xs_v, G)$ , states that value  $v_q$  represents the queue, which contains elements  $xs_v$
- $G \in \text{SeqQnames}$  is a collection of ghost names (depends on specific queue)
- Specification consists of three Hoare triples – one for each queue function
- **Important:**  $\text{isQueue}_S$  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**,  $\Psi$

## Definition (Concurrent Specification)

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

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

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

# 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**,  $\Psi$

## Definition (Concurrent Specification)

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

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

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

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

# 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**,  $\Psi$

## Definition (Concurrent Specification)

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

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

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

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

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

# 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**,  $\Psi$

## Definition (Concurrent Specification)

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

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

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

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

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

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

## 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**

$\gamma \Vdash_{\bullet} xS_v$   
Owned by queue

## **Fragmental view**

$\gamma \Vdash_{\circ} xS_v$   
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

$$\vdash \Vdash \exists \gamma. \gamma \Vdash_{\bullet} xs_v * \gamma \Vdash_{\circ} xs_v \quad (\text{Abstract State Alloc})$$

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

$$\gamma \Vdash_{\bullet} xs'_v * \gamma \Vdash_{\circ} xs_v \Rightarrow \gamma \Vdash_{\bullet} xs''_v * \gamma \Vdash_{\circ} xs''_v \quad (\text{Abstract State Update})$$

# HOCAP-style Specification

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

## Definition (HOCAP Specification)

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

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

# 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)

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

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

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

# HOCAP-style Specification

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

## Definition (HOCAP Specification)

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

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

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

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

# 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)

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

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

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

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

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

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

## Queue Client - A PoC Client

- Add two numbers after having two threads enqueue and subsequently dequeue them

```
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

- Add two numbers after having two threads enqueue and subsequently dequeue them
- Idea: a minimal client complex enough to require HOCAP-style specification

```
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

- Add two numbers after having two threads enqueue and subsequently dequeue them
- Idea: a minimal client complex enough to require HOCAP-style specification
- Uses parallel composition, so sequential specification insufficient

```
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

- Add two numbers after having two threads enqueue and subsequently dequeue them
- Idea: a minimal client complex enough to require HOCAP-style specification
- Uses parallel composition, so sequential specification insufficient
- Relies on dequeues not returning None, so concurrent specification insufficient

```
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

- Add two numbers after having two threads enqueue and subsequently dequeue them
- Idea: a minimal client complex enough to require HOCAP-style specification
- Uses parallel composition, so sequential specification insufficient
- Relies on dequeues not returning None, so concurrent specification insufficient
- HOCAP-style specification supports consistency and tracks 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)

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

## 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)

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

- When using the HOCAP-style Queue specification to prove the above, we will make  $P$  and  $Q$  talk about the tokens.
- E.g for enqueue:
  - $P = \text{TokA } Ga \vee \text{TokB } Ga$
  - $Q = \text{TokD1 } Ga \vee \text{TokD2 } Ga$

## Queue Specifications Overview

<b>Spec\Feature</b>	<b>Supports Tracking</b>	<b>Supports Concurrency</b>	
Sequential	✓	✗	
Concurrent	✗	✓	
HOCAP	✓	✓	

## Queue Specifications Overview

<b>Spec\Feature</b>	<b>Supports Tracking</b>	<b>Supports Concurrency</b>	<b>Price</b>
Sequential	✓	✗	199\$
Concurrent	✗	✓	249\$
HOCAP	✓	✓	399\$

## Queue Specifications Overview

<b>Spec\Feature</b>	<b>Supports Tracking</b>	<b>Supports Concurrency</b>	<b>Price</b>
Sequential	✓	✗	199\$
Concurrent	✗	✓	249\$
HOCAP	✓	✓	399\$

- HOCAP generalises Sequential and Concurrent Specs
- In fact, they are provably derivable from HOCAP

# HOCAP Derives Sequential

## Definition (Sequential Specification (reminder))

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

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

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

$\wedge \forall v_q, xs_v, G. \{\text{isQueues}(v_q, xs_v, G)\}$

$\text{ dequeue } v_q$

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

# HOCAP Derives Sequential

## Definition (HOCAP Specification)

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

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

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

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

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

■ Chose  $\text{isQueues}(v_q, x_{s_v}, G) = \text{isQueue}(v_q, G) * G.\gamma_{\text{Abst}} \mapsto_{\circ} x_{s_v}$

# HOCAP Derives Sequential

## Definition (HOCAP Specification)

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

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

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

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

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

- Chose  $\text{isQueues}(v_q, x_{s_v}, G) = \text{isQueue}(v_q, G) * G.\gamma_{\text{Abst}} \mapsto_{\circ} x_{s_v}$
- For enqueue, pick
  - $P = G.\gamma_{\text{Abst}} \mapsto_{\circ} x_{s_v}$
  - $Q = G.\gamma_{\text{Abst}} \mapsto_{\circ} v :: x_{s_v}$

# HOCAP Derives Sequential

## Definition (HOCAP Specification)

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

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

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

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

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

■ Chose  $\text{isQueues}(v_q, x_{s_v}, G) = \text{isQueue}(v_q, G) * G.\gamma_{\text{Abst}} \mapsto_{\circ} x_{s_v}$

■ For enqueue, pick

■  $P = G.\gamma_{\text{Abst}} \mapsto_{\circ} x_{s_v}$

■  $Q = G.\gamma_{\text{Abst}} \mapsto_{\circ} v :: x_{s_v}$

■ For dequeue, pick

■  $P = G.\gamma_{\text{Abst}} \mapsto_{\circ} x_{s_v}$

■  $Q = G.\gamma_{\text{Abst}} \mapsto_{\circ} [] \vee (\exists v, x'_{s_v}. x_{s_v} = x'_{s_v} ++ [v] * G.\gamma_{\text{Abst}} \mapsto_{\circ} x'_{s_v})$

## HOCAP Derives Concurrent

Left as an exercise :)

Thanks for your time

Questions?