Formal Verification of a Concurrent Bounded Queue in a Weak Memory Model

Glen Mével, Jacques-Henri Jourdan
ICFP 2021, online
LMF & Inria Paris
contribution:
spec and proof for a fine-grained concurrent queue
in the weak memory model of Multicore OCaml
**contribution:**
spec and proof for a fine-grained concurrent queue
in the weak memory model of Multicore OCaml

**this talk:**
specifying a concurrent data structure under weak memory
contribution:
spec and proof for a fine-grained concurrent queue
in the weak memory model of Multicore OCaml

this talk:
specifying a concurrent data structure under weak memory

specification challenges:

1. shared ownership $\Rightarrow$ logical atomicity
contribution:
spec and proof for a fine-grained concurrent queue in the weak memory model of Multicore OCaml

this talk:
specifying a concurrent data structure under weak memory

specification challenges:

1. shared ownership $\implies$ logical atomicity
2. weak memory $\implies$ thread synchronization
 contribution:
spec and proof for a fine-grained concurrent queue in the weak memory model of Multicore OCaml
	his talk:
specifying a concurrent data structure under weak memory

specification challenges:

1. shared ownership $\implies$ logical atomicity
2. weak memory $\implies$ thread synchronization
   - fine-grained concurrency $\implies$ weaker than lock-based
contribution: spec and proof for a fine-grained concurrent queue in the weak memory model of Multicore OCaml

this talk: specifying a concurrent data structure under weak memory

specification challenges:

1. shared ownership $\implies$ logical atomicity
2. weak memory $\implies$ thread synchronization
   - fine-grained concurrency $\implies$ weaker than lock-based

tool: Cosmo, our program logic for Multicore OCaml
Sequential queues
A specification for sequential queues

\[
\begin{align*}
\{ & \text{True} \} & & \{ & \text{IsQueue } q [v_0, \ldots, v_{n-1}] \} \\
\text{make } () & & \text{enqueue } q v \\
\{ & \lambda q. \text{IsQueue } q [] \} & & \{ & \lambda () . \text{IsQueue } q [v_0, \ldots, v_{n-1}, v] \} \\
\{ & \text{IsQueue } q [v_0, \ldots, v_{n-1}] \} & & \text{dequeue } q \\
\{ & \lambda v. 1 \leq n \times v = v_0 \times \text{IsQueue } q [v_1, \ldots, v_{n-1}] \}
\end{align*}
\]
A specification for sequential queues

\[
\begin{align*}
\{ \text{True} \} & \quad \{ \text{IsQueue } q [v_0, \ldots, v_{n-1}] \} \\
\text{make } () & \\
\{ \lambda q. \text{IsQueue } q [] \} & \\
\end{align*}
\]

\[
\begin{align*}
\{ \text{IsQueue } q [v_0, \ldots, v_{n-1}] \} & \\
\text{enqueue } q \; v & \\
\{ \lambda() . \text{IsQueue } q [v_0, \ldots, v_{n-1}, v] \} & \\
\end{align*}
\]

\[
\begin{align*}
\{ \text{IsQueue } q [v_0, \ldots, v_{n-1}] \} & \\
\text{dequeue } q & \\
\{ \lambda v . 1 \leq n \; * \; v = v_0 \; * \; \text{IsQueue } q [v_1, \ldots, v_{n-1}] \} & \\
\end{align*}
\]
A specification for sequential queues

\[
\begin{align*}
\{\text{True}\} & \quad \{\text{IsQueue} \ q \ [v_0, \ldots, v_{n-1}]\} \\
\text{make ()} & \quad \text{enqueue} \ q \ v \\
\{\lambda q. \text{IsQueue} \ q \ [\ ]\} & \quad \{\lambda () . \text{IsQueue} \ q \ [v_0, \ldots, v_{n-1}, v]\} \\
\text{dequeue} \ q & \\
\{\text{IsQueue} \ q \ [v_0, \ldots, v_{n-1}]\} & \quad \{\lambda v . 1 \leq n \ast v = v_0 \ast \text{IsQueue} \ q \ [v_1, \ldots, v_{n-1}]\}
\end{align*}
\]
A specification for sequential queues

\[
\begin{align*}
\{ \text{True} \} & \quad \{ \text{IsQueue } q [v_0, \ldots, v_{n-1}] \} \\
\text{make } () & \quad \text{enqueue } q \; v \\
\{ \lambda q. \text{IsQueue } q \; [\] \} & \quad \{ \lambda () . \text{IsQueue } q [v_0, \ldots, v_{n-1}, v] \} \\
\{ \text{IsQueue } q [v_0, \ldots, v_{n-1}] \} & \quad \text{dequeue } q \\
\{ \lambda v. 1 \leq n \; * \; v = v_0 \; * \; \text{IsQueue } q [v_1, \ldots, v_{n-1}] \} &
\end{align*}
\]
Concurrent queues
for now we assume sequential consistency: behaviors of the program are interleavings of its threads

can we keep the sequential spec?
for now we assume **sequential consistency**: behaviors of the program are interleavings of its threads

can we keep the sequential spec? valid, but...

```
IsQueue q [v_0, ..., v_{n-1}] is exclusive
⇒ effectively no concurrent usage
```
Invariants

[in a concurrent separation logic such as Iris]

an invariant holds at all times

idea: the user shares $q$ in an invariant:

$$I \triangleq \exists n, v_0, ..., v_{n-1}. \text{IsQueue } q [v_0, ..., v_{n-1}]$$

the invariant owns $q$
**Invariants**

[in a concurrent separation logic such as Iris]

an **invariant** holds at all times

idea: the user shares $q$ in an invariant:

\[ I \triangleq \exists n, v_0, ..., v_{n-1}. \text{IsQueue } q [v_0, ..., v_{n-1}] \]

the invariant owns $q$
[in a concurrent separation logic such as Iris]

an invariant holds at all times

idea: the user shares $q$ in an invariant:

$$I \triangleq \exists n, v_0, \ldots, v_{n-1}. \text{IsQueue} \ q \ [v_0, \ldots, v_{n-1}] \ast R \ [v_0, \ldots, v_{n-1}]$$

the invariant owns $q$
Invariants

[in a concurrent separation logic such as Iris]

an invariant holds at all times

idea: the user shares $q$ in an invariant:

$$I \triangleq \exists n, v_0, ..., v_{n-1}. \text{IsQueue } q [v_0, ..., v_{n-1}] \ast R [v_0, ..., v_{n-1}]$$

the invariant owns $q$

anyone can access $q$ by “opening” $I$:

$$\{P \ast I\} e \{I \ast Q\} \quad I \text{ is an invariant} \quad e \text{ completes in one step}$$

$$\{P\} e \{Q\}$$
Invariants

[in a concurrent separation logic such as Iris]

an **invariant** holds at all times

idea: the user shares \( q \) in an invariant:

\[
I \triangleq \exists n, v_0, ..., v_{n-1}. \text{IsQueue } q [v_0, ..., v_{n-1}] \ast R [v_0, ..., v_{n-1}]
\]

the invariant owns \( q \)

anyone can access \( q \) by “opening” \( I \):

\[
\{ P \ast I \} e \{ I \ast Q \} \quad I \text{ is an invariant} \quad e \text{ completes in one step}
\]

\[
\{ P \} e \{ Q \}
\]
Invariants

[in a concurrent separation logic such as Iris]

an invariant holds at all times

idea: the user shares \( q \) in an invariant:

\[
I \triangleq \exists n, v_0, ..., v_{n-1}. \text{IsQueue} \ q \ [v_0, ..., v_{n-1}] \ast R \ [v_0, ..., v_{n-1}]
\]

the invariant owns \( q \)

anyone can access \( q \) by “opening” \( I \):

\[
\{ P \ast I \} e \{ I \ast Q \} \quad I \text{ is an invariant} \quad e \text{ completes in one step}
\]

\[
\{ P \} e \{ Q \}
\]
Logical atomicity

[in Iris]

logically atomic triples are triples $\langle \cdot \rangle \cdot \langle \cdot \rangle$ such that:

\[
\begin{align*}
\langle P \rangle e \langle Q \rangle & \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \ Quad
Logical atomicity

[in Iris]

Logically atomic triples are triples $⟨·⟩ · ⟨·⟩$ such that:

- $⟨P⟩ e ⟨Q⟩$ implies $⟨P⟩ e ⟨Q⟩$
- $⟨P ∗ I⟩ e ⟨I ∗ Q⟩$ implies $⟨P⟩ e ⟨Q⟩$

$I$ is an invariant

tells that $e$ behaves "atomically"
Logical atomicity

[in Iris]

logically atomic triples are triples $\langle \cdot \rangle \cdot \langle \cdot \rangle$ such that:

\[
\langle P \rangle \mathbin{e} \langle Q \rangle \\
\{P\} \mathbin{e} \{Q\}
\]

\[
\langle P \mathbin{*} I \rangle \mathbin{e} \langle I \mathbin{*} Q \rangle \\
I \text{ is an invariant}
\]

\[
\langle P \rangle \mathbin{e} \langle Q \rangle
\]

tells that $e$ behaves “atomically”

intuition: $e$ takes a step which satisfies $\{P\} \cdot \{Q\}$

($\Longrightarrow$ related to linearizability)
Logical atomicity

[in Iris]

**logically atomic triples** are triples $\langle \cdot \rangle \cdot \langle \cdot \rangle$ such that:

\[
\begin{align*}
\langle x. P \rangle & e \langle Q \rangle \\
\forall x. \{ P \} & e \{ Q \}
\end{align*}
\]

\[
\begin{align*}
\langle x. P \ast I \rangle & e \langle I \ast Q \rangle \\
I & \text{ is an invariant}
\end{align*}
\]

\[
\begin{align*}
\langle x. P \rangle & e \langle Q \rangle
\end{align*}
\]

tells that $e$ behaves “atomically”

intuition: $e$ takes a step which satisfies $\forall x. \{ P \} \cdot \{ Q \}$

$(\Longrightarrow$ related to linearizability$)$

$x$ binds things which are known only during that step
A specification for concurrent queues under SC

\[
\begin{align*}
\{ \text{True} \} & \quad \langle n, v_0, \ldots, v_{n-1}. \text{IsQueue } q [v_0, \ldots, v_{n-1}] \rangle \\
\text{make } () & \quad \langle n, v_0, \ldots, v_{n-1}. \text{IsQueue } q [] \rangle \\
\lambda q. \text{IsQueue } q [] & \quad \langle \lambda () . \text{IsQueue } q [v_0, \ldots, v_{n-1}, v] \rangle \\
\end{align*}
\]

\[
\begin{align*}
\langle n, v_0, \ldots, v_{n-1}. \text{IsQueue } q [v_0, \ldots, v_{n-1}] \rangle & \quad \text{dequeue } q \\
\lambda v. 1 \leq n \land v = v_0 \land \text{IsQueue } q [v_1, \ldots, v_{n-1}] & \\
\end{align*}
\]
A specification for concurrent queues under SC

\{ \text{True} \}

\text{make} (\) \\
\{ \lambda q. \text{IsQueue} q [\] \}

\langle n, v_0, \ldots, v_{n-1}. \text{IsQueue} q [v_0, \ldots, v_{n-1}] \rangle

\text{enqueue} q \ v \\
\langle \lambda (). \text{IsQueue} q [v_0, \ldots, v_{n-1}, v] \rangle

\langle n, v_0, \ldots, v_{n-1}. \text{IsQueue} q [v_0, \ldots, v_{n-1}] \rangle

\text{dequeue} q \\
\langle \lambda v. 1 \leq n * \ v = v_0 * \text{IsQueue} q [v_1, \ldots, v_{n-1}] \rangle
A specification for concurrent queues under SC

\[
\begin{align*}
\{ \text{True} \} & \quad \langle n, v_0, \ldots, v_{n-1}. \text{IsQueue } q [v_0, \ldots, v_{n-1}] \rangle \\
\text{make } () & \quad \langle \lambda q. \text{IsQueue } q [] \rangle \\
\{ \lambda q. \text{IsQueue } q [] \} & \quad \langle \lambda () \cdot \text{IsQueue } q [v_0, \ldots, v_{n-1}, v] \rangle \\
\end{align*}
\]
A specification for concurrent queues under SC

\[
\begin{align*}
\{ \text{True} \} & \quad \langle n, v_0, \ldots, v_{n-1}. \text{IsQueue } q [v_0, \ldots, v_{n-1}] \rangle \\
\text{make } () & \quad \langle \lambda q. \text{IsQueue } q [] \rangle \\
\{ \lambda q. \text{IsQueue } q [] \} & \quad \langle n, v_0, \ldots, v_{n-1}. \text{IsQueue } q [v_0, \ldots, v_{n-1}] \rangle \\
\text{enqueue } q \ v & \quad \langle \lambda () . \text{IsQueue } q [v_0, \ldots, v_{n-1}, v] \rangle \\
\text{dequeue } q & \quad \langle n, v_0, \ldots, v_{n-1}. \text{IsQueue } q [v_0, \ldots, v_{n-1}] \rangle \\
& \quad \langle \lambda v. 1 \leq n \ * \ v = v_0 \ * \ \text{IsQueue } q [v_1, \ldots, v_{n-1}] \rangle \\
& \quad \text{(simplified)}
\end{align*}
\]
Concurrent queues in weak memory
Weak memory models:

- each thread has its own view of the state of the shared memory
  - example: C11
  - example: Multicore OCaml

[Dolan et al, PLDI 2018, *Bounding data races in space and time*]

operational semantics with thread-local views
Weak memory models

weak memory models: each thread has its own view of the state of the shared memory

- example: C11
- example: Multicore OCaml

[Dolan et al, PLDI 2018, *Bounding data races in space and time*

operational semantics with thread-local views
Weak memory models:

- each thread has its own **view** of the state of the shared memory

  - example: C11
  - example: Multicore OCaml

[Dolan et al, PLDI 2018, *Bounding data races in space and time*]

Operational semantics with thread-local views

**Cosmo:** a program logic for M-OCaml based on this semantics

[ICFP 2020]
based on Iris (hence: separation logic, ghost state, invariants)

assertions can be **subjective**: depend on current (thread’s) view

- example: $x \rightsquigarrow 42$
Cosmo

based on Iris (hence: separation logic, ghost state, invariants)

assertions can be **subjective**: depend on current (thread’s) view

- example: \( x \rightsquigarrow 42 \)

**restriction**: invariants are available to all threads

\( \implies \) **objective** assertions only
based on Iris (hence: separation logic, ghost state, **invariants**)

assertions can be **subjective**: depend on current (thread’s) view

- example: \( x \rightsquigarrow 42 \)

**restriction**: invariants are available to all threads

\( \implies \) **objective** assertions only

to be specified: IsQueue \( q [v_0, ..., v_{n-1}] \) is objective
Synchronizing through the queue?

can we keep the SC spec?
Synchronizing through the queue?

can we keep the SC spec? valid, usable in limited cases, but...

let enqueueer q =
  let x = array[2] in
  x[1] ← 3;
  { x[1] ⇝ 3 }
  enqueue q x

let dequeueer q =
  let x = dequeue q in
  { x[1] ⇝ 3 }
  do_something x[1]
Synchronizing through the queue?

can we keep the SC spec? valid, usable in limited cases, but...

let enqueuer q =
  | let x = array[2] in
  | x[1] ← 3 ;
  | { x[1] ↝ 3 }
  | enqueue q x

let dequeuer q =
  | let x = dequeue q in
  | { x[1] ↝ 3 }
  | do_something x[1]

x[1] ↝ 3 is subjective
⇒ cannot be transferred solely with an invariant
Synchronizing through the queue?

can we keep the SC spec? valid, usable in limited cases, but...

let enqueue q =
  let x = array[2] in
  x[1] ← 3 ;
  { x[1] ⇝ 3 }
  enqueue q x

let dequeue q =
  let x = dequeue q in
  { x[1] ⇝ 3 }
  do_something x[1]

x[1] ⇝ 3 is subjective
⇒ cannot be transferred solely with an invariant
to be specified: dequeue observes all writes done by enqueue
(⇒ “release-acquire” pattern)
Views in Cosmo

a lattice of views (larger = more up-to-date)
Views in Cosmo

a lattice of views (larger = more up-to-date)

new assertions:

\(
\uparrow V \quad \text{“the ambient view contains } V \text{”} \implies \text{subjective}
\)

\( P \odot V \quad \text{“} P \text{ where the ambient view has been fixed to } V \text{”} \implies \text{objective} \)
Views in Cosmo

a lattice of views (larger = more up-to-date)

new assertions:

\[ \uparrow \mathcal{V} \ "the \ ambient \ view \ contains \ \mathcal{V}" \Rightarrow \text{subjective} \]

\[ P @ \mathcal{V} \ "P \ where \ the \ ambient \ view \ has \ been \ fixed \ to \ \mathcal{V}" \Rightarrow \text{objective} \]

splitting rule:

\[ P \models \exists \mathcal{V}. (\uparrow \mathcal{V} \ast P @ \mathcal{V}) \]
Views in Cosmo

a lattice of views (larger = more up-to-date)

new assertions:

$\uparrow V$ “the ambient view contains $V$” $\implies$ subjective

$P @ V$ “$P$ where the ambient view has been fixed to $V$” $\implies$ objective

shareable via an invariant

splitting rule:

$P \models \exists V. (\uparrow V * P @ V)$
a lattice of views (larger = more up-to-date)

new assertions:

\( \uparrow V \) “the ambient view contains \( V \)” \( \Rightarrow \) subjective transferred via thread synchronization

\( P \otimes V \) “\( P \) where the ambient view has been fixed to \( V \)” \( \Rightarrow \) objective shareable via an invariant

splitting rule:

\[ P \models \exists V. (\uparrow V \ast P \otimes V) \]
idea: pretend the queue stores the views being transferred

\[ \text{IsQueue } q \ [ \ v_0 \ , \ldots , \ v_{n-1} \ ] \]

the enqueuer pushes its view alongside the enqueued value:

\[
\begin{align*}
\langle & n, \ v_0 \ , \ldots \ , \ v_{n-1} \ , \ \text{IsQueue } q \ [ \ v_0 \ , \ldots \ , \ v_{n-1} \ ] \\
\langle & \lambda(). \text{IsQueue } q \ [ \ v_0 \ , \ldots , \ v_{n-1} \ , \ v \ ] \rangle
\end{align*}
\]
Transferring views through the queue

idea: pretend the queue stores the views being transferred

\[ \text{IsQueue } q \ [(v_0, V_0), \ldots, (v_{n-1}, V_{n-1})] \]

the enqueuer pushes its view alongside the enqueued value:

\[
\begin{align*}
\langle n, v_0, \ldots, v_{n-1} \rangle \\
\text{IsQueue } q \ [(v_0, \ldots, v_{n-1})] \\
\text{enqueue } q \ v \\
\lambda(). \text{IsQueue } q \ [(v_0, \ldots, v_{n-1}, v)]
\end{align*}
\]
Transferring views through the queue

idea: pretend the queue stores the views being transferred

\[
\text{IsQueue } q \ [(v_0, \mathcal{V}_0), \ldots, (v_{n-1}, \mathcal{V}_{n-1})]\]

the enqueuer \textbf{pushes} its view alongside the enqueued value:

\[
\langle \lambda(). \text{IsQueue } q \ [(v_0, \mathcal{V}_0), \ldots, (v_{n-1}, \mathcal{V}_{n-1})] \rangle
\]

\[
\text{enqueue } q \ v
\]

\[
\langle \lambda(). \text{IsQueue } q \ [(v_0, \mathcal{V}_0), \ldots, (v_{n-1}, \mathcal{V}_{n-1}), (v, \mathcal{V})] \rangle
\]
Transferring views through the queue

idea: pretend the queue stores the views being transferred

\[
\text{IsQueue } q \ [(v_0, V_0), \ldots, (v_{n-1}, V_{n-1})]
\]

the dequeuer pulls that view:

\[
\langle n, v_0, \ldots, v_{n-1} \rangle
\]

\[
\text{IsQueue } q \ [v_0, v_1, \ldots, v_{n-1}]
\]

dequeue \( q \)

\[
\langle \lambda v. \text{IsQueue } q \ [v_1, \ldots, v_{n-1}] \rangle \quad * \ 1 \leq n \ * \ v = v_0 \]
Transferring views through the queue

idea: pretend the queue stores the views being transferred

\[
\text{IsQueue } q \ [(v_0, V_0), \ldots, (v_{n-1}, V_{n-1})]
\]

the dequeuer pulls that view:

\[
\langle n, (v_0, V_0), \ldots, (v_{n-1}, V_{n-1}) \rangle
\]

\[
\text{IsQueue } q \ [(v_0, V_0), (v_1, V_1), \ldots, (v_{n-1}, V_{n-1})]
\]

dequeue \( q \) \[
\langle \lambda v. \text{IsQueue } q \ [(v_1, V_1), \ldots, (v_{n-1}, V_{n-1})] \ * \uparrow V_0 \ * \ 1 \leq n \ * \ v = v_0 \rangle
\]
Comparison with refinement in weak memory

refinement spec: “this queue can replace a naïve queue + a lock”
Comparison with refinement in weak memory

refinement spec: “this queue can replace a naïve queue + a lock”

issue: induces synchronization between all operations

many lock-free queues do not (we try to avoid synchronizations!)

⇒ they do not satisfy the refinement spec
Comparison with refinement in weak memory

refinement spec: “this queue can replace a naïve queue + a lock”

issue: induces synchronization between all operations

many lock-free queues do not (we try to avoid synchronizations!)
⇒ they do not satisfy the refinement spec

our spec is weaker (no guaranteed sync. from dequeuer to enqueuer)
⇒ covers more lock-free queues
Conclusion
concurrent program verification:

- **invariants** share resources among threads
- **(logical) atomicity** is part of specs
Conclusion

concurrent program verification in weak memory:

- **invariants** share resources among threads
- **(logical) atomicity** is part of specs
- **view transfers** express synchronizations, also part of specs

Also in this work:

- proof of a non-trivial lock-free queue (does not refine a lock-based queue w.r.t. sync.)
- proof of a simple client
  - machine-checked (Coq, Iris)
Conclusion

concurrent program verification in weak memory:

- **invariants** share resources among threads
- (logical) **atomicity** is part of specs
- **view transfers** express synchronizations, also part of specs

also in this work:

- proof of a non-trivial lock-free queue
  (does not refine a lock-based queue w.r.t. sync.)

- proof of a simple client
- machine-checked (Coq, Iris) ✨
concurrent program verification in weak memory:

- **invariants** share resources among threads
- (logical) **atomicity** is part of specs
- **view transfers** express synchronizations, also part of specs

also in this work:

- proof of a non-trivial lock-free queue
  (does not refine a lock-based queue w.r.t. sync.)
  [a refinement proof in SC: Vindum & Birkedal, 2021, *Mechanized Verification of a Fine-Grained Concurrent Queue from Facebook’s Folly Library*]
- proof of a simple client
- machine-checked (Coq, Iris) 🍀