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

Glen Mével, Jacques-Henri Jourdan

ICFP 2021, online

LMF & Inria Paris

Introduction

contribution:

spec and proof for a fine-grained concurrent queue
in the weak memory model of Multicore OCaml

Introduction

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

Introduction

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 $\Longrightarrow$ logical atomicity

Introduction

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 $\Longrightarrow$ logical atomicity
2. weak memory $\Longrightarrow$ thread synchronization

Introduction

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 $\Longrightarrow$ logical atomicity
2. weak memory $\Longrightarrow$ thread synchronization
    - fine-grained concurrency $\Longrightarrow$ weaker than lock-based

Introduction

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

$$\left\{\text{True}\right\}\quad \texttt{make ()} \quad \left\{\lambda q.\ \text{IsQueue}\ q\ []\right\}$$

$$\left\{\text{IsQueue}\ q\ [v_0, \ldots, v_{n-1}]\right\}\quad \texttt{enqueue}\ q\ v \quad \left\{\lambda ().\ \text{IsQueue}\ q\ [v_0, \ldots, v_{n-1}, v]\right\}$$

$$\left\{\text{IsQueue}\ q\ [v_0, \ldots, v_{n-1}]\right\}\quad \texttt{dequeue}\ q \quad \left\{\lambda v.\ 1 \leq n \ * \ v = v_0 \ * \ \text{IsQueue}\ q\ [v_1, \ldots, v_{n-1}]\right\}$$

A specification for sequential queues

$$\left\{\text{True}\right\}\ \texttt{make ()}\ \left\{\lambda q.\,\text{IsQueue}\ q\ []\right\}$$

$$\left\{\text{IsQueue}\ q\ [v_0, \ldots, v_{n-1}]\right\}\ \texttt{enqueue}\ q\ v\ \left\{\lambda().\,\text{IsQueue}\ q\ [v_0, \ldots, v_{n-1}, v]\right\}$$

$$\left\{\text{IsQueue}\ q\ [v_0, \ldots, v_{n-1}]\right\}\ \texttt{dequeue}\ q\ \left\{\lambda v.\,1 \leq n\ *\ v = v_0\ *\ \text{IsQueue}\ q\ [v_1, \ldots, v_{n-1}]\right\}$$

A specification for sequential queues

$$\left\{\text{True}\right\}\ \texttt{make ()}\ \left\{\lambda q.\, \text{IsQueue}\ q\ []\right\}$$

$$\left\{\text{IsQueue}\ q\ [v_0, ..., v_{n-1}]\right\}\ \texttt{enqueue}\ q\ v\ \left\{\lambda ().\, \text{IsQueue}\ q\ [v_0, ..., v_{n-1}, v]\right\}$$

$$\left\{\text{IsQueue}\ q\ [v_0, ..., v_{n-1}]\right\}\ \texttt{dequeue}\ q\ \left\{\lambda v.\, 1 \leq n\ *\ v = v_0\ *\ \text{IsQueue}\ q\ [v_1, ..., v_{n-1}]\right\}$$

A specification for sequential queues

$$\left\{\text{True}\right\}\ \texttt{make ()}\ \left\{\lambda q.\,\text{IsQueue}\ q\ []\right\}$$

$$\left\{\text{IsQueue}\ q\ [v_0, \ldots, v_{n-1}]\right\}\ \texttt{enqueue}\ q\ v\ \left\{\lambda().\,\text{IsQueue}\ q\ [v_0, \ldots, v_{n-1}, v]\right\}$$

$$\left\{\text{IsQueue}\ q\ [v_0, \ldots, v_{n-1}]\right\}\ \texttt{dequeue}\ q\ \left\{\lambda v.\,1 \leq n \ * \ v = v_0 \ * \ \text{IsQueue}\ q\ [v_1, \ldots, v_{n-1}]\right\}$$

Concurrent queues

Concurrency

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

can we keep the sequential spec?

Concurrency

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
$\Longrightarrow$ 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:

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

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:

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

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:

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

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:

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

the invariant owns q

anyone can access q by "opening" I:

$$\frac{\{P * I\}\ e\ \{I * Q\} \qquad I \text{ is an invariant} \qquad e \text{ completes in one step}}{\{P\}\ e\ \{Q\}}$$

Invariants

[in a concurrent separation logic such as Iris]

an **invariant** holds at all times

idea: the user shares q in an invariant:

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

the invariant owns q

anyone can access q by "opening" I:

$$\frac{\{P * I\}\ e\ \{I * Q\} \qquad I \text{ is an invariant} \qquad e \text{ completes in one step}}{\{P\}\ e\ \{Q\}}$$

Invariants

[in a concurrent separation logic such as Iris]

an **invariant** holds at all times

idea: the user shares q in an invariant:

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

the invariant owns q

anyone can access q by "opening" I:

$$\frac{\{P * I\}\; e\; \{I * Q\} \qquad I \text{ is an invariant} \qquad e \text{ completes in one step}}{\{P\}\; e\; \{Q\}}$$

Logical atomicity

[in Iris]

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

$$\frac{\langle\quad P\rangle\, e\,\langle Q\rangle}{\{P\}\, e\,\{Q\}} \qquad\qquad \frac{\langle\quad P * I\rangle\, e\,\langle I * Q\rangle \qquad I \text{ is an invariant}}{\langle\quad P\rangle\, e\,\langle Q\rangle}$$

Logical atomicity

[in Iris]

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

$$\frac{\langle \quad P \rangle \; e \; \langle Q \rangle}{\{P\} \; e \; \{Q\}} \qquad\qquad \frac{\langle \quad P * I \rangle \; e \; \langle I * Q \rangle \qquad I \text{ is an invariant}}{\langle \quad P \rangle \; e \; \langle Q \rangle}$$

tells that e behaves "atomically"

Logical atomicity

[in Iris]

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

$$\frac{\langle \quad P \rangle \, e \, \langle Q \rangle}{\{P\} \, e \, \{Q\}} \qquad \frac{\langle \quad P * I \rangle \, e \, \langle I * Q \rangle \qquad I \text{ is an invariant}}{\langle \quad P \rangle \, e \, \langle Q \rangle}$$

tells that e behaves "atomically"

intuition: e takes a step which satisfies $\quad \{P\} \; \cdot \; \{Q\}$
($\Longrightarrow$ related to linearizability)

Logical atomicity

[in Iris]

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

$$\frac{\langle x.P\rangle\ e\ \langle Q\rangle}{\forall x.\ \{P\}\ e\ \{Q\}} \qquad\qquad \frac{\langle x.P * I\rangle\ e\ \langle I * Q\rangle \qquad I \text{ is an invariant}}{\langle x.P\rangle\ e\ \langle Q\rangle}$$

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

$$\left\{\text{True}\right\}\ \texttt{make}\ ()\ \left\{\lambda q.\ \text{IsQueue}\ q\ []\right\}$$

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

$$\left\langle n, v_0, \ldots, v_{n-1}.\ \text{IsQueue}\ q\ [v_0, \ldots, v_{n-1}]\right\rangle\ \texttt{dequeue}\ q\ \left\langle \lambda v.\ 1 \leq n\ *\ v = v_0\ *\ \text{IsQueue}\ q\ [v_1, \ldots, v_{n-1}]\right\rangle$$

A specification for concurrent queues under SC

$$\left\{\text{True}\right\}\ \texttt{make ()}\ \left\{\lambda q.\ \text{IsQueue}\ q\ []\right\}$$

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

$$\left\langle n, v_0, \ldots, v_{n-1}.\ \text{IsQueue}\ q\ [v_0, \ldots, v_{n-1}]\right\rangle\ \texttt{dequeue}\ q\ \left\langle \lambda v.\ 1 \leq n\ *\ v = v_0\ *\ \text{IsQueue}\ q\ [v_1, \ldots, v_{n-1}]\right\rangle$$

A specification for concurrent queues under SC

$$\left\{\text{True}\right\}\ \texttt{make ()}\ \left\{\lambda q.\ \text{IsQueue}\ q\ []\right\}$$

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

$$\left\langle n, v_0, \ldots, v_{n-1}.\ \text{IsQueue}\ q\ [v_0, \ldots, v_{n-1}]\right\rangle\ \texttt{dequeue}\ q\ \left\langle\lambda v.\ 1 \leq n\ *\ v = v_0\ *\ \text{IsQueue}\ q\ [v_1, \ldots, v_{n-1}]\right\rangle$$

A specification for concurrent queues under SC

$$\left\{\text{True}\right\}\ \texttt{make ()}\ \left\{\lambda q.\, \text{IsQueue}\ q\ []\right\}$$

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

$$\left\langle n, v_0, \ldots, v_{n-1}.\, \text{IsQueue}\ q\ [v_0, \ldots, v_{n-1}]\right\rangle\ \texttt{dequeue}\ q\ \left\langle \lambda v.\, 1 \leq n \ast v = v_0 \ast \text{IsQueue}\ q\ [v_1, \ldots, v_{n-1}]\right\rangle$$

(simplified)

Concurrent queues in weak memory

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

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

Cosmo: a program logic for M-OCaml based on this semantics
[ICFP 2020]

Cosmo

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
$\Longrightarrow$ **objective** assertions only

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
$\Longrightarrow$ **objective** assertions only

to be specified: IsQueue q $[v_0, \ldots, v_{n-1}]$ 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 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]
```

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]
```

$x[1] \rightsquigarrow 3$ is subjective

$\Longrightarrow$ cannot be transferred solely with an invariant

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]
```

$x[1] \rightsquigarrow 3$ is subjective

$\Longrightarrow$ cannot be transferred solely with an invariant

to be specified: dequeuer observes all writes done by enqueuer

($\Longrightarrow$ "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:

$\uparrow \mathcal{V}$ "the ambient view contains $\mathcal{V}$" $\Longrightarrow$ subjective

$P \mathbin{@} \mathcal{V}$ "P where the ambient view has been fixed to $\mathcal{V}$" $\Longrightarrow$ objective

Views in Cosmo

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

new assertions:

$\uparrow \mathcal{V}$ "the ambient view contains $\mathcal{V}$" $\implies$ subjective

$P \mathbin{@} \mathcal{V}$ "P where the ambient view has been fixed to $\mathcal{V}$" $\implies$ objective

splitting rule:

$$P \dashv\vdash \exists \mathcal{V}.\, (\uparrow \mathcal{V} * P \mathbin{@} \mathcal{V})$$

Views in Cosmo

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

new assertions:

$\uparrow\mathcal{V}$ "the ambient view contains $\mathcal{V}$" $\Longrightarrow$ subjective

$P \mathbin{@} \mathcal{V}$ "P where the ambient view has been fixed to $\mathcal{V}$" $\Longrightarrow$ objective
shareable via an invariant

splitting rule:

$$P \dashv\vdash \exists \mathcal{V}.\, (\uparrow\mathcal{V} * P \mathbin{@} \mathcal{V})$$

Views in Cosmo

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

new assertions:

$\uparrow\mathcal{V}$ "the ambient view contains $\mathcal{V}$" $\implies$ subjective
transferred via thread synchronization

$P @ \mathcal{V}$ "P where the ambient view has been fixed to $\mathcal{V}$" $\implies$ objective
shareable via an invariant

splitting rule:

$$P \dashv\vdash \exists \mathcal{V}. \, (\uparrow\mathcal{V} * P @ \mathcal{V})$$

Transferring views through the queue

idea: pretend the queue stores the views being transferred

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

the enqueuer pushes its view alongside the enqueued value:

$$\left\langle \begin{array}{l} n,\ v_0\ ,\ldots,\ v_{n-1}\ . \\ \quad \mathrm{IsQueue}\ q\ [\ v_0\ ,\ldots,\ v_{n-1}\ ] \end{array} \right\rangle$$

$$\texttt{enqueue}\ q\ v$$

$$\left\langle \lambda().\ \mathrm{IsQueue}\ q\ [\ v_0\ ,\ldots,\ v_{n-1}\ ,\ v\ ] \right\rangle$$

Transferring views through the queue

idea: pretend the queue stores the views being transferred

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

the enqueuer pushes its view alongside the enqueued value:

$$\left\langle \begin{array}{l} n,\ v_0 \quad , \ldots,\ v_{n-1} \quad . \\ \quad \mathsf{IsQueue}\ q\ [\ v_0 \quad , \ldots,\ v_{n-1} \quad] \end{array} \right\rangle$$

$$\texttt{enqueue}\ q\ v$$

$$\left\langle \lambda().\, \mathsf{IsQueue}\ q\ [\ v_0 \quad , \ldots,\ v_{n-1} \quad ,\ v \quad] \right\rangle$$

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 pushes its view alongside the enqueued value:

$$\left\langle \begin{array}{l} n, (v_0, \mathcal{V}_0), \ldots, (v_{n-1}, \mathcal{V}_{n-1}). \\ \qquad \text{IsQueue}\ q\ [(v_0, \mathcal{V}_0), \ldots, (v_{n-1}, \mathcal{V}_{n-1})] \qquad * \ \uparrow\mathcal{V} \end{array} \right\rangle$$

$$\texttt{enqueue}\ q\ v$$

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

Transferring views through the queue

idea: pretend the queue stores the views being transferred

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

the dequeuer pulls that view:

$$\left\langle \begin{array}{l} n, \ v_0 \qquad , \dots, \ v_{n-1} \qquad . \\ \quad \text{IsQueue } q \ [\ v_0 \qquad , \ v_1 \qquad , \dots, \ v_{n-1} \qquad] \end{array} \right\rangle$$

$$\texttt{dequeue } q$$

$$\left\langle \lambda v.\, \text{IsQueue } q \ [\ v_1 \qquad , \dots, \ v_{n-1} \qquad] \qquad * \ 1 \leq n \ * \ v = v_0 \right\rangle$$

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 dequeuer pulls that view:

$$\left\langle \begin{array}{l} n, (v_0, \mathcal{V}_0), \ldots, (v_{n-1}, \mathcal{V}_{n-1}). \\ \quad \text{IsQueue}\ q\ [(v_0, \mathcal{V}_0), (v_1, \mathcal{V}_1), \ldots, (v_{n-1}, \mathcal{V}_{n-1})] \end{array} \right\rangle$$

$$\texttt{dequeue}\ q$$

$$\left\langle \lambda v.\, \text{IsQueue}\ q\ [(v_1, \mathcal{V}_1), \ldots, (v_{n-1}, \mathcal{V}_{n-1})] \ * \ \uparrow\mathcal{V}_0 \ * \ 1 \leq n \ * \ v = v_0 \right\rangle$$

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!)
$\Longrightarrow$ 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!)
$\Longrightarrow$ they do not satisfy the refinement spec

our spec is weaker (no guaranteed sync. from dequeuer to enqueuer)
$\Longrightarrow$ covers more lock-free queues

Conclusion

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

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.)
  [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)