Verifying Wait-Freedom for Concurrent Higher-Order Programs

ECOOP'26

Egor Namakonov, Lars Birkedal, Amin Timany

Aarhus University

June 8, 2026

Wait-freedom: the strongest progress guarantee

Wait-freedom: the strongest progress guarantee

> *"An [implementation of] operation [on a concurrent data structure] is wait-free if it guarantees that every call finishes its execution in a finite number of steps."*
> *Herlihy and Shavit (2012)*

Wait-freedom: the strongest progress guarantee

> *"An [implementation of] operation [on a concurrent data structure] is wait-free if it guarantees that every call finishes its execution in a finite number of steps."*
> *Herlihy and Shavit (2012)*

```
(* Does the increment in the left thread always terminate? *)
cnt.incr () || while true do cnt.incr (); done
```

Wait-freedom: the strongest progress guarantee

> *"An [implementation of] operation [on a concurrent data structure] is wait-free if it guarantees that every call finishes its execution in a finite number of steps."*
> *Herlihy and Shavit (2012)*

```
(* Does the increment in the left thread always terminate? *)
cnt.incr () || while true do cnt.incr (); done
```

```
let new_cnt_blocking () =
  let c = ref 0 in
  let l = new_lock () in
  let i _ =
    acquire l;
    let n = !c in
    c := n + 1;
    release l;
    n
  in { incr = i }
```

✗

Wait-freedom: the strongest progress guarantee

> *"An [implementation of] operation [on a concurrent data structure] is wait-free if it guarantees that every call finishes its execution in a finite number of steps."*
> *Herlihy and Shavit (2012)*

```
(* Does the increment in the left thread always terminate? *)
cnt.incr () || while true do cnt.incr (); done
```

```
let new_cnt_blocking () =
  let c = ref 0 in
  let l = new_lock () in
  let i _ =
    acquire l;
    let n = !c in
    c := n + 1;
    release l;
    n
  in { incr = i }
```

✗

```
let new_cnt_lockfree () =
  let c = ref 0 in
  let rec i _ =
    let n = !c in
    if CAS(c, n, n + 1)
    then n
    else i ()
  in { incr = i }
```

✗

Wait-freedom: the strongest progress guarantee

> *"An [implementation of] operation [on a concurrent data structure] is wait-free if it guarantees that every call finishes its execution in a finite number of steps."*
> *Herlihy and Shavit (2012)*

```
(* Does the increment in the left thread always terminate? *)
cnt.incr () || while true do cnt.incr (); done
```

```
let new_cnt_blocking () =
  let c = ref 0 in
  let l = new_lock () in
  let i _ =
    acquire l;
    let n = !c in
    c := n + 1;
    release l;
    n
  in { incr = i }
```

✗

```
let new_cnt_lockfree () =
  let c = ref 0 in
  let rec i _ =
    let n = !c in
    if CAS(c, n, n + 1)
    then n
    else i ()
  in { incr = i }
```

✗

```
let new_cnt_waitfree () =
  let c = ref 0 in
  let i _ = FAA(c, 1)
  in { incr = i }
```

✓

Wait-free implementations are the hardest

Blocking queue, Vindum and Birkedal (2021)

```
let cg_enqueue lock head x =
  let rec enqueue_go x head =
    match head with
    | None
-> Some (x, None)
    | Some p -> Some (fst p, enqueue_go x (snd p))
  in
  acquire lock;
  head := enqueue_go x !head;
  release lock

let cg_dequeue lock head () =
  acquire lock;
  let v = match !head with
          | None
-> None
          | Some p -> head := snd p; Some (fst p)
  in
  release lock;
  v

let cg_queue () =
  let head = ref None in
  let lock = newlock () in
  (cg_dequeue lock head, cg_enqueue lock head x)
```

Wait-free implementations are the hardest

Blocking queue, Vindum and Birkedal (2021)

```
let cg_enqueue lock head x =
  let rec enqueue_go x head =
    match head with
    | None
-> Some (x, None)
    | Some p -> Some (fst p, enqueue_go x (snd p))
  in
  acquire lock;
  head := enqueue_go x !head;
  release lock

let cg_dequeue lock head () =
  acquire lock;
  let v = match !head with
          | None
-> None
          | Some p -> head := snd p; Some (fst p)
  in
  release lock;
  v

let cg_queue () =
  let head = ref None in
  let lock = newlock () in
  (cg_dequeue lock head, cg_enqueue lock head x)
```

Lock-free Michael-Scott queue, version by Vindum and Birkedal (2021)

```
let get_value x =
  match x with
  | None
-> let rec spin () = spin () in spin ()
  | Some v -> v

let ms_dequeue toSent toTail =
  let rec loop () =
    let head  = !toSent in
    let tail  = !toTail in
    let toNext = snd (get_value !head) in
    let p     = NewProph in
    let next  = !toNext in
    if head = (Resolve (!toSent) p ()) then
      if head = tail then
        match !next with
        | None   -> None
        | Some _ -> CAS toTail tail next; loop ()
        end
      else
        if CAS toSent head next then
          Some (get_value (fst (get_value !next)))
        else loop ()
    else loop ()
  in loop ()

let ms_enqueue toTail x =
  let node = ref (Some (Some x, ref (ref None))) in
  let rec loop () =
    let tail  = !toTail in
    let toNext = snd (get_value !tail) in
    let next  = !toNext in
    if tail = !toTail then
      match !next with
      | None   ->
        if CAS toNext next node then
          (CAS toTail tail node; ())
        else loop ()
      | Some _ -> CAS toTail tail next; loop ()
      end
    else loop ()
  in loop ()

let ms_queue () =
  let node
= ref (Some (None, ref (ref None))) in
  let toTail = ref node in
  let toSent = ref node in
  ((ms_dequeue toSent toTail), (ms_enqueue toTail))
```

Wait-free implementations are the hardest

Blocking queue, Vindum and Birkedal (2021)

```
let cg_enqueue lock head x =
  let rec enqueue_go x head =
    match head with
    | None
-> Some (x, None)
    | Some p -> Some (fst p, enqueue_go x (snd p))
  in
  acquire lock;
  head := enqueue_go x !head;
  release lock

let cg_dequeue lock head () =
  acquire lock;
  let v = match !head with
          | None
-> None
          | Some p -> head := snd p; Some (fst p)
  in
  release lock;
  v

let cg_queue () =
  let head = ref None in
  let lock = newlock () in
  (cg_dequeue lock head, cg_enqueue lock head x)
```

Lock-free Michael-Scott queue, version by Vindum and Birkedal (2021)

```
let get_value x =
  match x with
  | None
-> let rec spin () = spin () in spin ()
  | Some v -> v

let ms_dequeue toSent toTail =
  let rec loop () =
    let head  = !toSent in
    let tail  = !toTail in
    let toNext = snd (get_value !head) in
    let p     = NewProph in
    let next  = !toNext in
    if head = (Resolve (!toSent) p ()) then
      if head = tail then
        match !next with
        | None   -> None
        | Some _ -> CAS toTail tail next; loop ()
        end
      else
        if CAS toSent head next then
          Some (get_value (fst (get_value !next)))
        else loop ()
    else loop ()
  in loop ()

let ms_enqueue toTail x =
  let node = ref (Some (Some x, ref (ref None))) in
  let rec loop () =
    let tail  = !toTail in
    let toNext = snd (get_value !tail) in
    let next  = !toNext in
    if tail = !toTail then
      match !next with
      | None   ->
        if CAS toNext next node then
          (CAS toTail tail node; ())
        else loop ()
      | Some _ -> CAS toTail tail next; loop ()
      end
    else loop ()
  in loop ()

let ms_queue () =
  let node
= ref (Some (None, ref (ref None))) in
  let toTail = ref node in
  let toSent = ref node in
  ((ms_dequeue toSent toTail), (ms_enqueue toTail))
```

Wait-free queue, Kogan and Petrank (2011) (translated w/ Claude)

```
type node
= { mutable value: int; mutable next: node option ref;
                mutable enqTid: int; mutable deqTid: int }
type op_desc = { phase: int; pending: bool; enqueue: bool; node: node option }

let head
= ref (mk_sentinel ())
let tail
= ref (mk_sentinel ())
let state = Array.init num_thrds
    (fun _ -> { phase = -1; pending = false; enqueue = true; node = None })

let max_phase () =
  Array.fold_left (fun m d -> max m d.phase) (-1) state

let is_still_pending tid ph =
  state.(tid).pending && state.(tid).phase <= ph

let help phase =
  Array.iteri (fun i d ->
    if d.pending && d.phase <= phase then
      if d.enqueue then help_enq i phase
      else
help_deq i phase
    ) state

let enq value =
  let phase = max_phase () + 1 in
  state.(tid) <- { phase; pending = true; enqueue = true;
                   node = Some { value; next = ref None;
                                 enqTid = tid; deqTid = -1 } };

  help phase;
  help_finish_enq ()

let help_enq tid phase =
  while is_still_pending tid phase do
    let last = !tail in
    let next = !(last.next) in
    if last = !tail then
      match next with
      | None ->
        if is_still_pending tid phase then
          if CAS last.next None (state.(tid).node) then
            help_finish_enq ()
      | Some _ ->
        help_finish_enq ()
  done

let help_finish_enq () =
  let last = !tail in
  match !(last.next) with
  | None -> ()
  | Some next ->
      CAS state.(tid) cur_desc { cur_desc with pending = false };
      CAS tail last next
    end
let deq () =
  let phase = max_phase () + 1 in
  state.(tid) <- { phase; pending = true; enqueue = false; node
  help phase;
  help_finish_deq ();
  match state.(tid).node with
  | None     -> raise Empty
  | Some node -> node.next.value

let help_deq tid phase =
  while is_still_pending tid phase do
    let first = !head in
    let last  = !tail in
    let next
= !(first.next) in
    if first = !head then
      if first = last then
        match next with
        | None ->
          let cur_desc = state.(tid) in
          if last = !tail && is_still_pending tid phase then
            CAS state.(tid) cur_desc
                { cur_desc with pending = false; node = None }
        | Some _ -> help_finish_enq ()
      else
        let cur_desc = state.(tid) in
        let node
= cur_desc.node in
        if is_still_pending tid phase then begin
          if first = !head && node <> Some first then
            if not (CAS state.(tid) cur_desc
                    { cur_desc with node = Some first }) then
          else begin
            CAS first.deqTid (-1) tid;
            help_finish_deq ()
          end
        end
  done

let help_finish_deq () =
  let first = !head in
  match !(first.next) with
  | None -> ()
  | Some next ->
    let tid = first.deqTid in
    if tid <> -1 then begin
      let cur_desc = state.(tid) in
      if first = !head then begin
        CAS state.(tid) cur_desc { cur_desc with pending = fals
```

Wait-freedom is hard to even define

"An operation is wait-free if it guarantees that every call finishes its execution in a finite number of steps." [Herlihy and Shavit (2012)]

Wait-freedom is hard to even define

Is the data structure initialized?

What if operation was stored in heap?

How to define formally in an ML-like language?

"An operation is wait-free if it guarantees that every call finishes its execution in a finite number of steps." [Herlihy and Shavit (2012)]

What about safety?

Who calls it? Is it an arbitrary client program?

How the execution is scheduled?

Wait-freedom is hard to even define

Is the data structure initialized?

What if operation was stored in heap?

How to define formally in an ML-like language?

"An operation is wait-free if it guarantees that every call finishes its execution in a finite number of steps." [Herlihy and Shavit (2012)]

What about safety?

Who calls it? Is it an arbitrary client program?

How the execution is scheduled?

$$waitFree(\mathsf{op}) : Prop$$

Wait-freedom is hard to even define

Is the data structure initialized?

What if operation was stored in heap?

How to define formally in an ML-like language?

"An operation is wait-free if it guarantees that every call finishes its execution in a finite number of steps." [Herlihy and Shavit (2012)]

What about safety?

Who calls it? Is it an arbitrary client program?

How the execution is scheduled?

$$waitFree_{\mathfrak{C}}(\mathsf{op}) : Prop$$

need to account for initialization; omit this parameter for simplicity

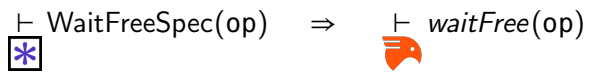
Wait-freedom is hard to even define, harder to prove.

$\vdash \mathit{waitFree}(\mathsf{op})$

Wait-freedom is hard to even define, harder to prove. Solution: verify the operation in Iris instead

$\vdash \mathsf{WaitFreeSpec}(\mathsf{op})$ $\Rightarrow$ $\vdash \mathit{waitFree}(\mathsf{op})$

Wait-freedom is hard to even define, harder to prove. Solution: verify the operation in Iris instead



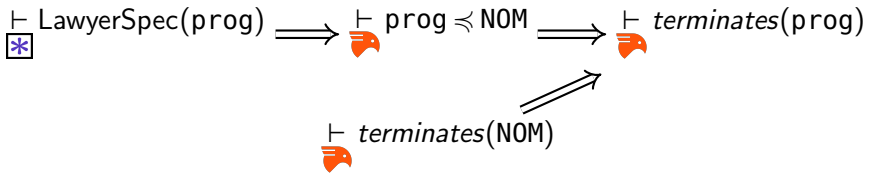
Our contributions, all mechanized in Rocq:

- Definition *waitFree*(op) of op's wait-freedom in higher-order setting;
- Iris-based specification WaitFreeSpec(op) and the corresponding adequacy theorem;
- Case studies for simple algorithms;
- Adapting the approach for verifying realistic, *not exactly wait-free* algorithms

Iris Workshop'25: termination proofs with Lawyer logic

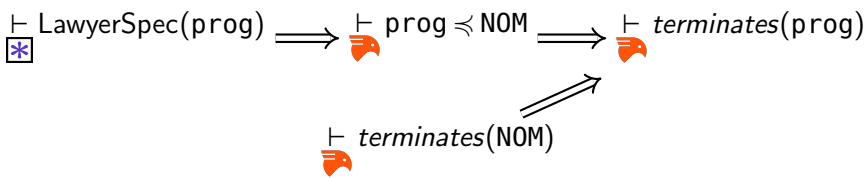
Iris Workshop'25: termination proofs with Lawyer logic

Lawyer approach, restricted to non-blocking concurrency:



Iris Workshop'25: termination proofs with Lawyer logic

Lawyer approach, restricted to non-blocking concurrency:



LTS	Language	NOM ("No-Obligations Model")
State	thread pool and heap	finite *fuel* assigned to each thread
Labels	thread ids	thread ids
Transitions	small-step semantics	besides others, *burning fuel on every step*

Not done yet: specs imply weaker property for open programs

Not done yet: specs imply weaker property for open programs

Key property for refining closed `prog` to `NOM`

$\vdash_{\boxed{*}}$ LawyerSpec(`prog`) $\Longrightarrow$ $\vdash_{\boxed{*}}$ execution of `prog` preserves invariants and refinement to `NOM`

Not done yet: specs imply weaker property for open programs

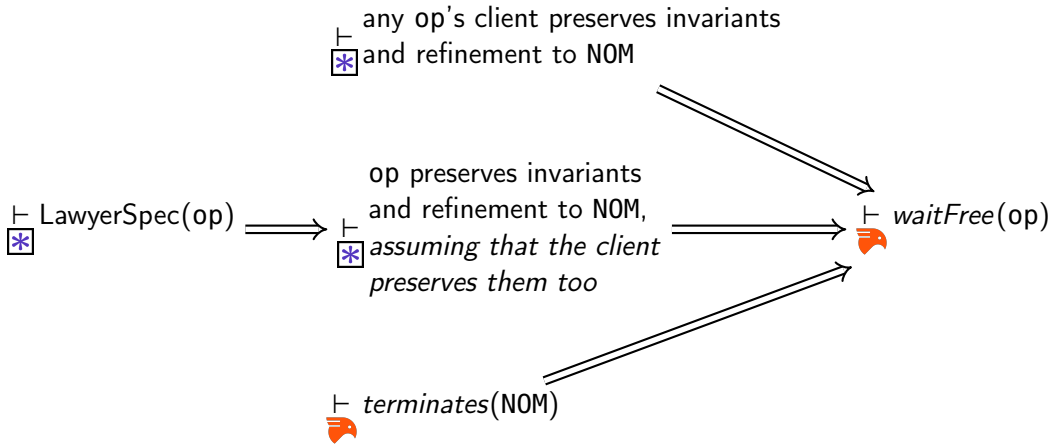
Key property for refining closed `prog` to `NOM`

$\vdash_{\boxed{*}}$ LawyerSpec(`prog`) $\Longrightarrow$ $\vdash_{\boxed{*}}$ execution of `prog` preserves invariants and refinement to `NOM`

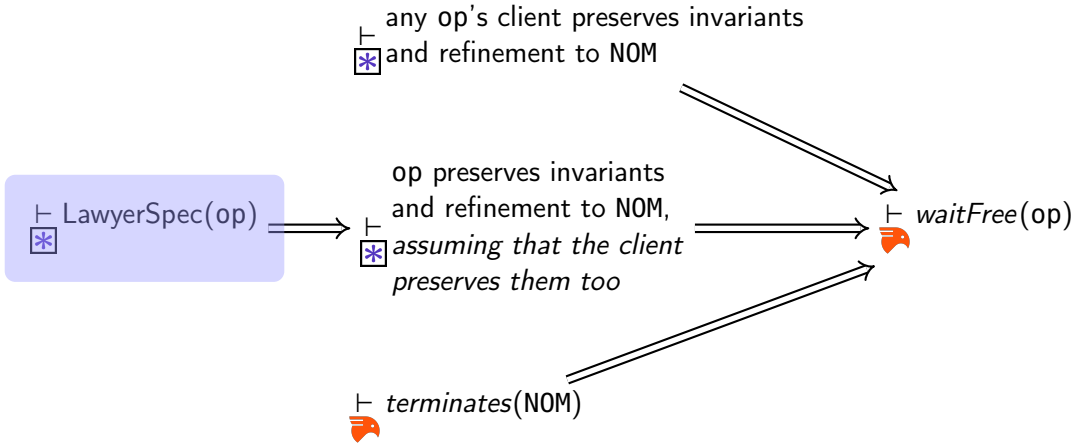
turns into a weaker property for standalone wait-free `op`:

$\vdash_{\boxed{*}}$ LawyerSpec(`op`) $\Longrightarrow$ $\vdash_{\boxed{*}}$ execution of `op` preserves invariants and refinement to `NOM`, *assuming that client of* `op` *preserves them too*

User proves spec and applies our new adequacy theorem



User proves spec and applies our new adequacy theorem



Reusing Lawyer logic to prove wait-freedom specification

```
let incr x =

    FAA(c, 1)
```

Reusing Lawyer logic to prove wait-freedom specification

Key part of proving $\vdash$ LawyerSpec(`incr`): $\boxed{*}$

For every τ : Thread
and x : *Val*:

$\{ \text{DB}_\tau * \text{DB}_\tau * \boxed{\exists n : \mathbb{Z}.\, c \mapsto n}^{\mathcal{N}} \}^{\tau}$

```
let incr x =

    FAA(c, 1)
```

{True}

Reusing Lawyer logic to prove wait-freedom specification

Key part of proving $\vdash$ LawyerSpec(`incr`): $\boxed{*}$

For every τ : Thread
and x : *Val*:

$\{\boxed{}_\tau * \boxed{}_\tau * \boxed{\exists n : \mathbb{Z}.\, c \mapsto n}^{\mathcal{N}}\}^\tau$

```
let incr x =

    FAA(c, 1)
```

{True}

Amount of fuel and concrete invariant depend on operation

"phase" resource omitted for simplicity

Reusing Lawyer logic to prove wait-freedom specification

Key part of proving $\vdash$ LawyerSpec(`incr`): $\boxed{*}$

For every τ : Thread
and x : *Val*:

$\{ \text{DB}_\tau * \text{DB}_\tau * \boxed{\exists n : \mathbb{Z}.\, c \mapsto n}^{\mathcal{N}} \}^{\tau}$

```
let incr x =

    FAA(c, 1)
```

{True}

Reusing Lawyer logic to prove wait-freedom specification

Key part of proving $\vdash$ LawyerSpec(`incr`):

$\boxed{*}$

For every τ : Thread
and x : *Val*:

$\{\exists_\tau * \exists_\tau * \boxed{\exists n : \mathbb{Z}.\, c \mapsto n}^{\mathcal{N}}\}^\tau$

```
let incr x =
```

$\{\exists_\tau\}$

```
    FAA(c, 1)
```

{True}

{True}

$$\textsc{Lawyer-beta-fuel}\ \frac{\{P\}\ e[v/x]\ \{Q\}^\tau_{\mathcal{E}}}{\{\exists_\tau * P\}\ (\lambda x.\ e)\, v\ \{Q\}^\tau_{\mathcal{E}}}$$

Reusing Lawyer logic to prove wait-freedom specification

Key part of proving $\vdash$ LawyerSpec(`incr`): $\boxed{*}$

For every τ : Thread
and x : *Val*:

$\{\boxed{\exists n : \mathbb{Z}.\, c \mapsto n}^{\mathcal{N}}\}^{\tau}$
let incr x =
  $\{\}_{\tau}$
    $\{\}_{\tau}^{\top \setminus \mathcal{N}}$
    FAA(c, 1)
    $\{\exists n.\, c \mapsto n\}^{\top \setminus \mathcal{N}}$
  {True}
{True}

LAWYER-BETA-FUEL

$$\frac{\{P\}\ e[v/x]\ \{Q\}^{\tau}_{\mathcal{E}}}{\{\}_{\tau} * P\}\ (\lambda x.\ e)\, v\ \{Q\}^{\tau}_{\mathcal{E}}}$$

IRIS-INV-ACC-TL

$$\frac{\{P * Q\}\ e\ \{P * R\}^{\tau}_{\mathcal{E} \setminus \mathcal{N}} \quad e \text{ is atomic} \quad \text{timeless}(P) \quad \mathcal{N} \subseteq \mathcal{E}}{\left\{\boxed{P}^{\mathcal{N}} * Q\right\}\ e\ \{R\}^{\tau}_{\mathcal{E}}}$$

Reusing Lawyer logic to prove wait-freedom specification

Key part of proving $\vdash$ LawyerSpec(`incr`): $\boxed{*}$

For every τ : Thread
and x : *Val*:

$\{\exists_\tau * \exists_\tau * \boxed{\exists n : \mathbb{Z}.\, c \mapsto n}^{\mathcal{N}}\}^{\tau}$
let incr x =
  $\{\exists_\tau\}$
    $\{\exists_\tau * \exists n.\, c \mapsto n\}^{\top \setminus \mathcal{N}}$
    FAA(c, 1)
    $\{\exists n.\, c \mapsto n\}^{\top \setminus \mathcal{N}}$
  {True}
{True}

LAWYER-BETA-FUEL

$$\frac{\{P\}\ e[v/x]\ \{Q\}^{\tau}_{\mathcal{E}}}{\{\exists_\tau * P\}\ (\lambda x.\ e)\, v\ \{Q\}^{\tau}_{\mathcal{E}}}$$

IRIS-INV-ACC-TL

$$\frac{\{P * Q\}\ e\ \{P * R\}^{\tau}_{\mathcal{E}\setminus\mathcal{N}} \quad e \text{ is atomic} \quad \text{timeless}(P) \quad \mathcal{N} \subseteq \mathcal{E}}{\left\{\boxed{P}^{\mathcal{N}} * Q\right\}\ e\ \{R\}^{\tau}_{\mathcal{E}}}$$

LAWYER-FAA-FUEL

$$\{\exists_\tau * \ell \mapsto n\}\ \mathsf{FAA}(\ell, k)\ \{m.\, m = n * \ell \mapsto (n+k)\}^{\tau}_{\mathcal{E}}$$

Reusing Lawyer logic to prove wait-freedom specification

Key part of proving $\vdash$ LawyerSpec(`incr`): $\boxed{*}$

For every τ : Thread
and x : *Val*:

$\{\boxed{\exists n : \mathbb{Z}.\, c \mapsto n}^{\mathcal{N}}\}^{\tau}$
let incr x =
  $\{\boxed{}_{\tau}\}$
    $\{\boxed{}_{\tau} * \exists n.\, c \mapsto n\}^{\top \setminus \mathcal{N}}$
    FAA(c, 1)
    $\{\exists n.\, c \mapsto n\}^{\top \setminus \mathcal{N}}$
  {True}
{True}

✓

LAWYER-BETA-FUEL

$$\frac{\{P\}\ e[v/x]\ \{Q\}^{\tau}_{\mathcal{E}}}{\{\boxed{}_{\tau} * P\}\ (\lambda x.\ e)\, v\ \{Q\}^{\tau}_{\mathcal{E}}}$$

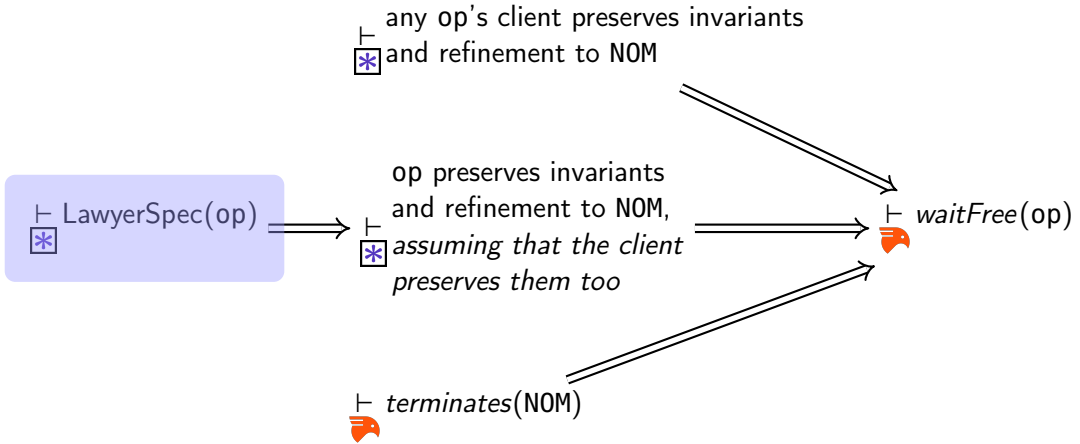
IRIS-INV-ACC-TL

$$\frac{\{P * Q\}\ e\ \{P * R\}^{\tau}_{\mathcal{E}\setminus\mathcal{N}} \quad e \text{ is atomic} \quad \text{timeless}(P) \quad \mathcal{N} \subseteq \mathcal{E}}{\left\{\boxed{P}^{\mathcal{N}} * Q\right\}\ e\ \{R\}^{\tau}_{\mathcal{E}}}$$

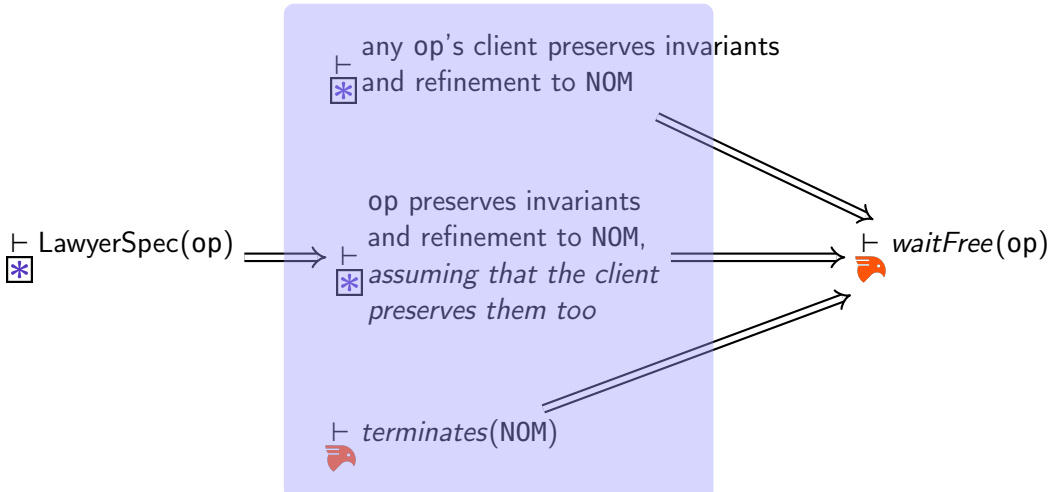
LAWYER-FAA-FUEL

$$\{\boxed{}_{\tau} * \ell \mapsto n\}\ \mathsf{FAA}(\ell, k)\ \{m.\, m = n * \ell \mapsto (n+k)\}^{\tau}_{\mathcal{E}}$$

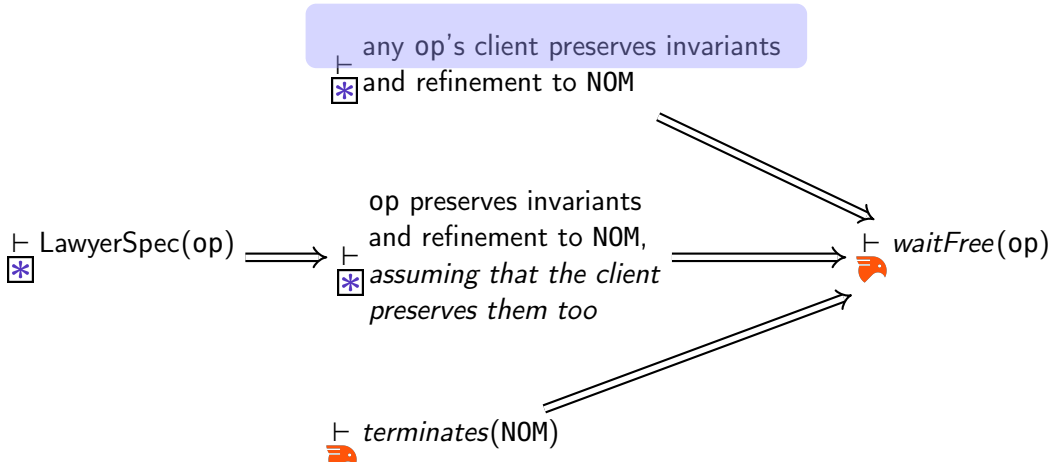
User proves spec and applies our new adequacy theorem



User proves spec and applies our new adequacy theorem



User proves spec and applies our new adequacy theorem



Invariants of the wait-free operation must be preserved

E.g. safe termination of `let incr x = FAA(c, 1)`
relies on $\boxed{\exists n : \mathbb{Z}.\, c \mapsto n}$ being preserved by its client

Invariants of the wait-free operation must be preserved, *i.e.* the operation should be "robustly safe"

E.g. safe termination of `let incr x = FAA(c, 1)`
relies on $\boxed{\exists n : \mathbb{Z}.\, c \mapsto n}$ being preserved by its client

op's client

Robust safety of verified `op`, more formally:

$\vdash \forall \texttt{prog} : \mathit{Expr}.\ \mathsf{LawyerSpec}(\texttt{op}) \mathrel{-\!\!*} \mathsf{PresInv}(\texttt{prog}[\texttt{op}/x])$

$\boxed{*}$

Invariants of the wait-free operation must be preserved, *i.e.* the operation should be "robustly safe"

E.g. safe termination of `let incr x = FAA(c, 1)`
relies on $\boxed{\exists n : \mathbb{Z}.\, c \mapsto n}$ being preserved by its client

op's client

Robust safety of verified `op`, more formally:

$\vdash \forall \texttt{prog} : \mathit{Expr}.\ \mathrm{LawyerSpec}(\texttt{op}) \mathrel{-\!\!*} \mathrm{PresInv}(\texttt{prog}[\texttt{op}/x])$

$\boxed{*}$

How to prove it for any `prog`?

In fact, most of clients preserve operation's invariants

What preserves the invariant $\boxed{\exists n : \mathbb{Z}.\, c \mapsto n}$ of `incr`?

In fact, most of clients preserve operation's invariants

What preserves the invariant $\boxed{\exists n : \mathbb{Z}.\, c \mapsto n}$ of `incr`?

In fact, most of clients preserve operation's invariants

What preserves the invariant $\boxed{\exists n : \mathbb{Z}.\, c \mapsto n}$ of `incr`?

In fact, most of clients preserve operation's invariants

```
incr ()
```

What preserves the invariant $\boxed{\exists n : \mathbb{Z}.\, c \mapsto n}$ of `incr`?

Calling the verified operation ✓

In fact, most of clients preserve operation's invariants

```
                  ||  let n = 5 + 5 in  ||
incr ()           ||  incr n;           ||    incr ()
```

What preserves the invariant $\boxed{\exists n : \mathbb{Z}.\, c \mapsto n}$ of `incr`?

Calling the verified operation (+ in multiple threads, with any argument) ✓

In fact, most of clients preserve operation's invariants

```
                ||  let n = 5 + 5 in  ||  while true do
incr ()         ||  incr n;           ||    incr ()
                ||                    ||  done
```

What preserves the invariant $\boxed{\exists n : \mathbb{Z}.\, c \mapsto n}$ of `incr`?

- Calling the verified operation (+ in multiple threads, with any argument) ✓
- Running infinitely ✓

In fact, most of clients preserve operation's invariants

```
                    ||            ||  let n = 5 + 5 in  ||  while true do
                    ||  incr ()   ||  incr n;           ||    incr ()
() ()               ||            ||                    ||  done
```

What preserves the invariant $\boxed{\exists n : \mathbb{Z}.\, c \mapsto n}$ of `incr`?

- Calling the verified operation (+ in multiple threads, with any argument) ✓
- Running infinitely ✓
- Getting stuck outside of the operation ✓

In fact, most of clients preserve operation's invariants

```
let l = ref 0 in  ||           ||  let n = 5 + 5 in  ||  while true do
l := true;        ||  incr ()  ||  incr n;           ||    incr ()
() ()             ||           ||                    ||  done
```

What preserves the invariant $\boxed{\exists n : \mathbb{Z}.\, c \mapsto n}$ of `incr`?

- Calling the verified operation (+ in multiple threads, with any argument) ✓
- Running infinitely ✓
- Getting stuck outside of the operation ✓
- Writing to *fresh* locations ✓

In fact, most of clients preserve operation's invariants

```
let l = ref 0 in  ||           ||  let n = 5 + 5 in  ||  while true do
l := true;        ||  incr ()  ||  incr n;           ||    incr ()
() ()             ||           ||  [n] := true       ||  done
```

What preserves the invariant $\boxed{\exists n : \mathbb{Z}.\, c \mapsto n}$ of `incr`?

- Calling the verified operation (+ in multiple threads, with any argument) ✓
- Running infinitely ✓
- Getting stuck outside of the operation ✓
- Writing to *fresh* locations ✓
- Writing to *arbitrary* locations ✗

Prove robust safety with logical relations

Prove robust safety with logical relations

$\mathrm{RS_E}(\texttt{prog} : \mathit{Expr}) \approx$

execution of `prog` preserves invariants,
might get stuck,
and the returned value (if any) is "safe"

$\mathrm{RS_V}(v : \mathit{Val}) \approx$

v can be used in safe programs
(*e.g.* location allocated by the client)

Prove robust safety with logical relations

$\mathsf{RS}_\mathsf{E}(\texttt{prog} : \mathit{Expr}) \approx$

execution of `prog` preserves invariants,
might get stuck,
and the returned value (if any) is "safe"

$\mathsf{RS}_\mathsf{V}(v : \mathit{Val}) \approx$

v can be used in safe programs
(*e.g.* location allocated by the client)

Break this circularity by defining RS_E and RS_V with more primitive Iris propositions!

Prove robust safety with logical relations

$\mathrm{RS}_{\mathrm{E}}(\texttt{prog} : \mathit{Expr}) \approx$

execution of `prog` preserves invariants,
might get stuck,
and the returned value (if any) is "safe"

$\mathrm{RS}_{\mathrm{V}}(v : \mathit{Val}) \approx$

v can be used in safe programs
(*e.g.* location allocated by the client)

Break this circularity by defining RS_{E} and RS_{V} with more primitive Iris propositions!
With that, show a general property:

$$\vdash \forall \texttt{prog}, \vec{v}.\ \mathit{NoPtrArith}(\texttt{prog}) \mathrel{-\!\!*} \mathrm{RS}_{\mathrm{V}}(\vec{v}) \mathrel{-\!\!*} \mathrm{RS}_{\mathrm{E}}(\texttt{prog}[\vec{v} / \overrightarrow{\mathit{freeVars}(\texttt{prog})}])$$

$\boxed{*}$

Prove robust safety with logical relations

$\mathrm{RS_E}(\texttt{prog} : \mathit{Expr}) \approx$

execution of `prog` preserves invariants,
might get stuck,
and the returned value (if any) is "safe"

$\mathrm{RS_V}(v : \mathit{Val}) \approx$

v can be used in safe programs
(*e.g.* location allocated by the client)

Break this circularity by defining $\mathrm{RS_E}$ and $\mathrm{RS_V}$ with more primitive Iris propositions!
With that, show a general property:

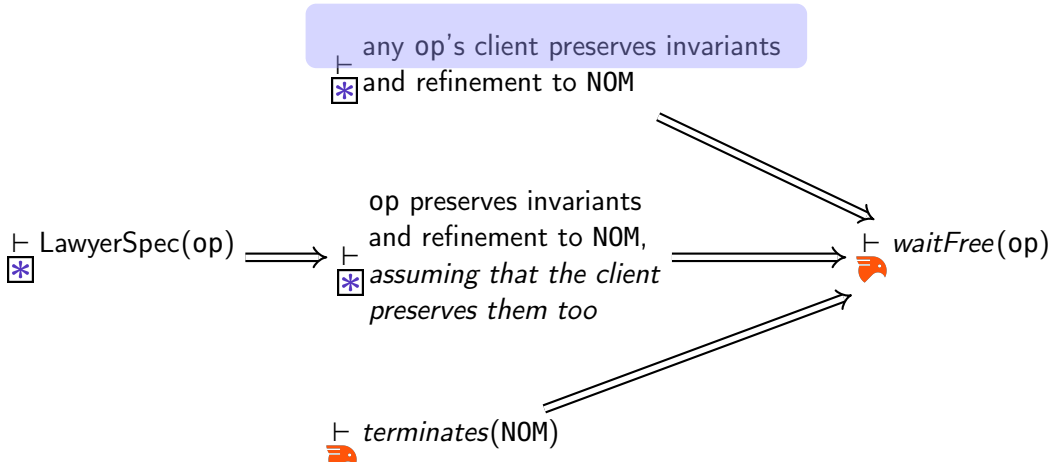
$$\vdash \boxed{*}\ \forall \texttt{prog}, \vec{v}.\ \mathit{NoPtrArith}(\texttt{prog}) \mathrel{-\!\!*} \mathrm{RS_V}(\vec{v}) \mathrel{-\!\!*} \mathrm{RS_E}(\texttt{prog}[\vec{v}/\overrightarrow{\mathit{freeVars}(\texttt{prog})}])$$

and conclude robust safety of verified `op`:

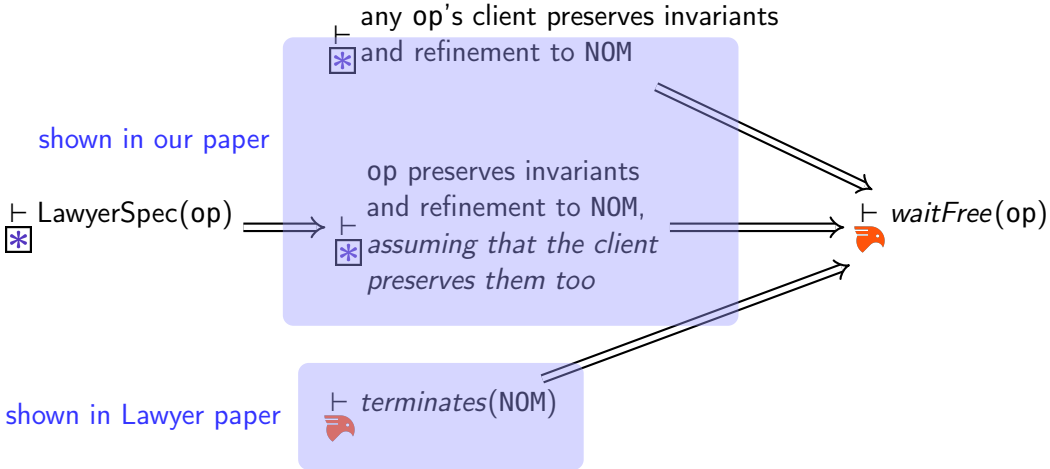
$$\vdash \boxed{*}\ \forall \texttt{prog}.\ \mathit{NoPtrArith}(\texttt{prog}) \mathrel{-\!\!*} \mathrm{RS_V}(\texttt{op}) \mathrel{-\!\!*} \mathrm{RS_E}(\texttt{prog}[\texttt{op}/x])$$

by adapting the proof of LawyerSpec(`op`)

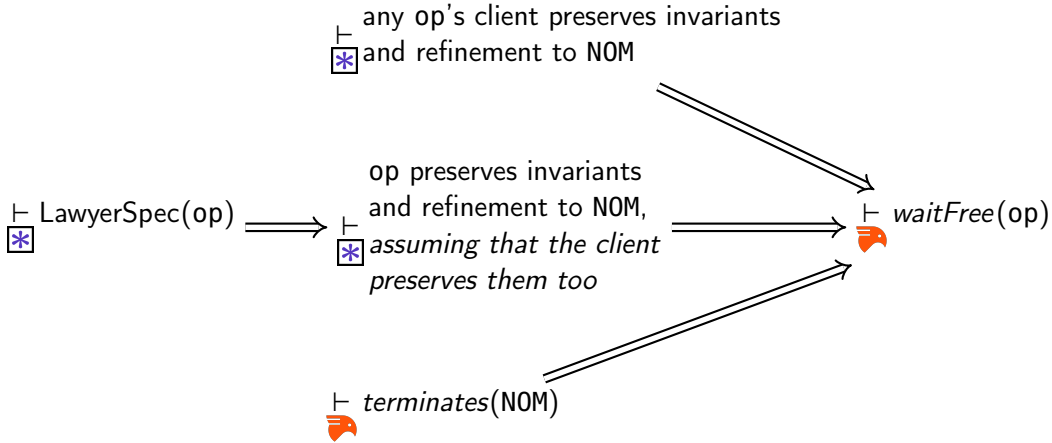
User proves spec and applies our new adequacy theorem



User proves spec and applies our new adequacy theorem



User proves spec and applies our new adequacy theorem



More complicated algorithms are not textbook-wait-free

More complicated algorithms are not textbook-wait-free

- Wait-free operation might get stuck if it actually inspects the input argument (*e.g.*, `list_map f l` with wait-free `f` when `l` is not a list)

More complicated algorithms are not textbook-wait-free

- Wait-free operation might get stuck if it actually inspects the input argument (*e.g.*, `list_map f l` with wait-free `f` when `l` is not a list)
- Realistic wait-free data structures restrict concurrency ...

Realistic wait-free data structures restrict concurrency

```
void enq(int value) {
  long phase = maxPhase() + 1;
  state.set(TID, new
    OpDesc(phase, true, true, new Node(value, TID)));
  help(phase);
  help_finish_enq();
}
```

1. Post its own operation
2. Help previously posted operations
3. Complete its own operation

```
AtomicReferenceArray<OpDesc> state;

WFQueue () {
  state = new AtomicReferenceArray<OpDesc>(NUM_THRDS);
  ...
```

Adapting our approach to restricted wait-freedom

Adapting our approach to restricted wait-freedom

Result for 1-enqueuer, 1-dequeuer queue by Jayanti and Petrovic (2005):

$\vdash$ *waitFreeRestr*([enqueuer, dequeuer])

Adapting our approach to restricted wait-freedom

Result for 1-enqueuer, 1-dequeuer queue by Jayanti and Petrovic (2005):

$\vdash$ *waitFreeRestr*([enqueuer, dequeuer])

$$\mathit{waitFreeRestr}(\mathsf{ops} : \mathit{List}\,\mathit{Val}) \approx \begin{array}{l}\text{similar to } \mathit{waitFree}(\mathsf{ops}[i]) \text{ for every } i,\\ \text{but } i\text{th thread can only call } \mathsf{ops}[i];\\ \text{forks are prohibited, but } \mathsf{ops}[i] = \mathsf{ops}[j] \text{ is allowed}\end{array}$$

Adapting our approach to restricted wait-freedom

Result for 1-enqueuer, 1-dequeuer queue by Jayanti and Petrovic (2005):

$\vdash$ *waitFreeRestr*([enqueuer, dequeuer])

waitFreeRestr(ops : *List Val*) $\approx$ similar to *waitFree*(ops[i]) for every i, but ith thread can only call ops[i]; forks are prohibited, but ops[i] = ops[j] is allowed

WaitFreeRestrSpec(ops) $\approx$ for each op $\in$ ops, show $\forall a, \tau.\ \{\Diamond_\tau * \ldots * \Diamond_\tau * \mathrm{tok}_{\mathrm{op}}\}\ \mathrm{op}\ a\ \{\mathrm{tok}_{\mathrm{op}}\}^\tau$ assuming there are only *count*(op, ops) many $\mathrm{tok}_{\mathrm{op}}$

$\vdash$ WaitFreeRestrSpec(ops) $\Rightarrow$ $\vdash$ *waitFreeRestr*(ops)

Adapting our approach to restricted wait-freedom

Result for 1-enqueuer, 1-dequeuer queue by Jayanti and Petrovic (2005):

$\vdash$ *waitFreeRestr*([enqueuer, dequeuer])

$$\mathit{waitFreeRestr}(\mathsf{ops} : \mathit{List}\,\mathit{Val}) \approx \begin{array}{l}\text{similar to } \mathit{waitFree}(\mathsf{ops}[i]) \text{ for every } i,\\ \text{but } i\text{th thread can only call } \mathsf{ops}[i];\\ \text{forks are prohibited, but } \mathsf{ops}[i] = \mathsf{ops}[j] \text{ is allowed}\end{array}$$

$$\mathsf{WaitFreeRestrSpec}(\mathsf{ops}) \approx \text{for each } \mathsf{op} \in \mathsf{ops}, \text{ show } \forall a, \tau.\ \{\boxminus_\tau * \ldots * \boxminus_\tau * \mathsf{tok}_{\mathsf{op}}\}\ \mathsf{op}\, a\ \{\mathsf{tok}_{\mathsf{op}}\}^\tau$$

assuming there are only $\mathit{count}(\mathsf{op}, \mathsf{ops})$ many $\mathsf{tok}_{\mathsf{op}}$

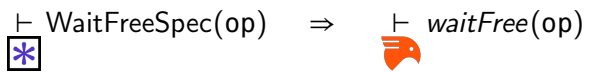
$\vdash \mathsf{WaitFreeRestrSpec}(\mathsf{ops}) \quad \Rightarrow \quad \vdash \mathit{waitFreeRestr}(\mathsf{ops})$

Logical relation and adequacy proof change, but the program logic stays the same!

Future work

- Linearizability in the same framework (requires prophecy variables in Trillium)
- Verification of universal constructions for wait-freedom
- Minor changes, *e.g.*, being more flexible with amount of fuel

Wait-freedom is hard to even define, harder to prove. Solution: verify the operation in Iris instead



Our contributions, all mechanized in Rocq:

- Definition *waitFree*(op) of op's wait-freedom in higher-order setting;
- Iris-based specification WaitFreeSpec(op) and the corresponding adequacy theorem;
- Case studies for simple algorithms;
- Adapting the approach for verifying realistic, *not exactly wait-free* algorithms

Wait-freedom is hard to even define, harder to prove. Solution: verify the operation in Iris instead

$$\underset{\boxed{*}}{\vdash \mathrm{WaitFreeSpec}(\mathsf{op})} \quad \Rightarrow \quad \underset{\text{🦅}}{\vdash \mathit{waitFree}(\mathsf{op})}$$

Our contributions, all mechanized in Rocq:

- Definition *waitFree*(op) of op's wait-freedom in higher-order setting;
- Iris-based specification WaitFreeSpec(op) and the corresponding adequacy theorem;
- Case studies for simple algorithms;
- Adapting the approach for verifying realistic, *not exactly wait-free* algorithms

Thank you!

Backup slides

Logical relation

$$\mathsf{RS_E}(e) \triangleq \forall \tau.\ \{\mathsf{True}\}^{\lightning}\ e\ \{v.\mathsf{RS_V}(v)\}^{\tau}$$

$$\mathsf{RS_V}(()) \triangleq \mathsf{RS_V}(b : \mathbb{B}) \triangleq \ldots \triangleq \mathsf{True}$$

$$\ldots$$

$$\mathsf{RS_V}((v_1, v_2)) \triangleq \mathsf{RS_V}(v_1) * \mathsf{RS_V}(v_2)$$

$$\mathsf{RS_V}(\ell : \mathit{Loc}) \triangleq \boxed{(\exists v.\ \ell \mapsto v * \mathsf{RS_V}(v)) \vee \mathsf{freed}(\ell)}$$

$$\mathsf{RS_V}(\lambda\, x.e) \triangleq \forall \tau, v.\ \{\mathsf{RS_V}(v)\}^{\lightning}\ (\lambda\, x.e)v\ \{r.\mathsf{RS_V}(r)\}^{\tau}$$

The $\lightning$ triples are different from those used in Lawyer proofs:

- they don't enforce model refinement, *i.e.* only consider safety
- they allow to get stuck

Bibliography

- Herlihy, Maurice and Nir Shavit (2012). *The Art of Multiprocessor Programming, Revised Reprint*.
- Jayanti, Prasad and Srdjan Petrovic (2005). "Logarithmic-time single deleter, multiple inserter wait-free queues and stacks". In: DOI: 10.1007/11590156_33.
- Kogan, Alex and Erez Petrank (2011). "Wait-free queues with multiple enqueuers and dequeuers". In: DOI: 10.1145/1941553.1941585.
- Namakonov, Egor, Lars Birkedal, and Amin Timany (June 2026). "Verifying wait-freedom for concurrent higher-order programs". In: DOI: doi.org/10.4230/LIPIcs.ECOOP.2026.30.
- Namakonov, Egor, Justus Fasse, et al. (Apr. 2026). "Lawyer: Modular Obligations-Based Liveness Reasoning in Higher-Order Impredicative Concurrent Separation Logic". In: DOI: 10.1145/3798240.
- Vindum, Simon Friis and Lars Birkedal (2021). "Contextual refinement of the Michael-Scott queue (proof pearl)". In: DOI: 10.1145/3437992.3439930.