# Verifying Wait-Freedom for Concurrent Higher-Order Programs

**Egor Namakonov**
Aarhus University, Denmark

**Lars Birkedal**
Aarhus University, Denmark

**Amin Timany**
Aarhus University, Denmark

## Abstract

Wait-freedom is the strongest non-blocking progress guarantee for concurrent data structures, ensuring that every operation completes in a finite number of steps regardless of interference from other threads. While verification of wait-freedom has been studied for first-order languages, verifying it for higher-order programming languages with general references remains an open challenge. In such languages, operations may be used by arbitrary, unverified higher-order clients, making it unclear how to even define wait-freedom formally in terms of programs' semantics, let alone prove it.

In this paper, we present the first framework for verifying wait-freedom of concurrent programs written in a higher-order language with general references. Our approach is based on the Lawyer concurrent separation logic which has been recently introduced for termination verification. We identify a specification pattern in the Lawyer logic that captures wait-freedom. To establish this connection formally, we prove a novel adequacy theorem for Lawyer which states that programs which are proven correct in the Lawyer logic against a specification in the aforementioned specification pattern are wait-free. Proving wait-freedom requires to show that all calls made to operations by any arbitrary client terminate. Thus, as a part of proving the adequacy theorem above, we need to prove that the behavior of the client of the data structure is safe in the sense that it does not break the internal invariants of the data structure, *e.g.*, by directly manipulating the data structure's internal state. To this end, we develop a logical relations model that establishes safety for all clients once and for all.

We demonstrate the effectiveness of our approach by proving wait-freedom for several representative examples, including a higher-order list map function, and a memory-efficient single-producer, single-consumer queue. For the latter, wait-freedom is conditional in that, as the name suggests, there can be at most one enqueuer thread and one dequeuer thread. To capture this formally, we introduce the notion of restricted wait-freedom as a variant of wait-freedom that restricts the number of concurrent threads, and show how our approach can support reasoning about restricted wait-freedom. All our results have been mechanized on top of the Rocq Prover and using the Iris separation logic framework that Lawyer is also based on.

**2012 ACM Subject Classification** Software and its engineering → Formal software verification; Theory of computation → Logic and verification; Theory of computation → Concurrency

**Keywords and phrases** separation logic, higher-order logic, concurrency, formal verification

**Digital Object Identifier** 10.4230/LIPIcs.ECOOP.2026.20

**Related Version** *Full Version*: `https://zenodo.org/records/19607892`

**Supplementary Material** *Software (Artifact)*: `https://zenodo.org/records/19607892` [29]
*Software (ECOOP 2026 Artifact Evaluation approved artifact)*:
`https://doi.org/10.4230/DARTS.12.1.21`

**Funding** This work was supported in part by a Villum Investigator grant (VIL73403) and Center for Basic Research in Program Verification (CPV, 25804), from Villum Fonden. This work was co-funded by the European Union (ERC, CHORDS, 101096090). Views and opinions expressed are



however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

**Acknowledgements** We thank the anonymous reviewers for the feedback that helped to make our work more accessible, and Justus Fasse for the discussions on the specification style.

## 1 Introduction

Any implementation of a concurrent data structure must account for possible interference between multiple operations on that data structure running concurrently. Done wrongly, it might lead to incorrect behavior of operations, or even worse, to undefined behavior of an entire program. The simplest way to ensure concurrent operations' correctness is to resort to *blocking concurrency*, *e.g.*, using a lock, allowing only one operation to progress at any given moment, and forcing others to wait for its completion. However, in many settings, *e.g.*, in operating system kernels [4], blocking concurrency is discouraged as it might lead to poor performance, and due to risk of deadlocking the entire system. Therefore, in such cases a *lock-free* implementation is more desirable. Lock-free data structure implementations guarantee that among the concurrently running operations, at least one of them must run to completion. Yet, lock-freedom does not guarantee that there is no starvation. That is, a specific call to a lock-free data structure's operation might not terminate because it is starved by other calls. To avoid starvation, one must use so-called *wait-free* [15] implementations. Wait-freedom is the strongest form of non-blocking concurrency. It guarantees that *every* individual call to the data structure operations must run to completion regardless of how many other operations of the same data structure run concurrently with it.

We illustrate the differences between blocking, lock-free, and wait-free concurrency using a simple and contrived example: a concurrent counter implementation featuring an increment operation (and a read operation which we elide). Figure 1 shows multiple possible implementations of the increment operation, all of which store the current value of the counter under the location $l$. The implementation in Figure 1a allows only one thread at a time to access the counter, forcing all others to wait until the lock is released. The lock-free implementation (Figure 1b), implemented via an atomic Compare-And-Set (CAS) operation in a loop, does not wait for other concurrent operations on the counter to complete. It attempts to increment using the CAS operation, and if that fails, it retries. This implementation is lock-free: if the CAS operation fails, then the counter value must have changed after it has been read on the line above the CAS operation, which in turn means that another increment operation must have succeeded. However, this lock-free implementation can starve threads, even under some fair schedulers. (Even though unlikely, a pathological but fair scheduler could schedule the thread, and subsequently preempt it right before the CAS operation, only to return to it, every time, after another increment has succeeded.) Finally, we have the wait-free implementation in Figure 1c. It uses the atomic Fetch-And-Add (FAA) operation which atomically increments the counter. Thus, the increment always finishes in a single, atomic step. In our discussions throughout the rest of the paper we will refer to this wait-free implementation of increment. To this end, we introduce a wrapper function `incr`, so that we can treat `incr` $l$ as a function (that ignores its argument and executes increment).

In this contrived example of a simple concurrent counter (Figure 1c) we essentially relegate the problem of making the data structure wait-free to the hardware using the FAA operation. In reality, as we will see below, for data structures ever so slightly more complex than this wait-free implementations are far from trivial to implement and to formally reason about.

```
(* m is a lock instance *)
let incr_block l =
  acquire m;
  let n = !l in
  l := n + 1;
  release m;
  n
```

**(a)** Blocking implementation.

```
let rec incr_lockfree l =
  let n = !l in
  if CAS(l, n, n + 1) then
    n
  else
    incr_lockfree l
```

**(b)** Lock-free, non-wait-free implementation.

```
let incr_waitfree l =
  FAA(l, 1)

let incr l () =
  incr_waitfree l
```

**(c)** Wait-free implementation and a wrapper for it.

**Figure 1** Comparison of counter increment implementations; we use an OCaml-like syntax. Location $l$ stores an integer.

```
let simple_op () =
  (* Could be allocated at address 0x42. *)
  let b = ref true in
  let rec loop () =
    if !b then ()
    else loop ()
  (* Should terminate immediately. *)
  in loop ()
```

```
(* Parallel composition is written as || *)
let good_client =
  let r := ref simple_op in
  simple_op () || !r ()

(* Direct memory access is written as [·] *)
let bad_client =
  [0x42] := false || simple_op ()
```

**Figure 2** An example of a wait-free operation together with two possible clients.

Indeed, such implementations must be designed to execute concurrently in a way that neither do they have to wait for the completion of other operations, nor are they impeded in a way by the other operations that forces them to retry. Therefore, a wait-free implementation of a data structure is often much more complicated than a lock-free or blocking implementation of the same data structure (see *e.g.* wait-free queue of Kogan and Petrank [23] versus the lock-free queue of Michael and Scott [28]). Thus, it is often also more challenging to verify.

In this paper, we show how to use a higher-order concurrent program logic to verify wait-freedom for programs written in a concurrent *higher-order* programming language with general references. Such languages achieve greater modularity by supporting features such as closures and first-class modules. Our work goes significantly beyond the state of the art [8, 26] which has only considered wait-freedom of programs in first-order programming languages without general references. Working with higher-order programs, and with general references, introduces many new challenges that one needs to answer. In fact, it is not even clear a priori how to formally define wait-freedom in our setting. Informally, it is often explained that a wait-free operation should be able to make progress in any program context under any scheduler. However, it is not entirely clear here what one means by "any program context", in particular in our higher-order setting. A "context" (or *client*) of an operation can be any program that calls that operation, either directly or indirectly (through a reference), possibly concurrently – see *e.g.*, the `good_client` in Figure 2. However, it does not make sense to consider literally any client. In particular, the client's behavior should be "safe" in that it should not be able to violate data encapsulation of the wait-free data structure, *i.e.*, it should not be able to directly accessing data structure's internal state. For a simple example of what can go wrong see the `bad_client` in Figure 2 which directly writes to `0x42` while concurrently running `simple_op`. The example `bad_client` could potentially violate an invariant of `simple_op` (namely the invariant that the local state variable `b` is always true) and thus break the wait-freedom of `simple_op`.

The example `bad_client` in Figure 2 shows that the progress guarantee of a wait-free operation crucially relies on its client being safe. Indeed, prior works on wait-freedom verification for first-order programs enforce safety of clients in one way or another, but their approaches do not apply in our setting. One way to enforce clients' safety is by using a programming language that syntactically separates accesses to the operation's and client's states [26]. This approach, however, is not viable for the higher-order language we consider. (In general, the introduction of higher-order store enables behaviors not present in a first-order setting, including turning a terminating program into non-terminating, as demonstrated by [9]). Another approach to ensuring safety of clients is by requiring them to be verified [8, 32]. In addition to not being applicable in our higher-order setting, this approach also inherently limits the applicability of the wait-freedom result: now every single client of potential interest must be verified in the corresponding framework. Rather, we take an approach that does not require one to verify individual clients but instead establishes operations' progress within arbitrary clients. This also includes those clients out of reach of existing solutions, *e.g.*, higher-order clients.

In this work, we present the first solution to the verification of wait-freedom for a realistic, higher-order language. Our solution is based on the recent Lawyer logic [31], an extension of the Trillium [35] and Iris [22] frameworks. So far, Lawyer has only been used for establishing (fair) termination of closed programs. The key insight of our work is that a specific form of specifications given to a function forces that the function to terminate *in any context, without either blocking or starving*. That is, such Lawyer specifications essentially codify, for all intents and purposes, wait-freedom in the Lawyer program logic. However, there are two main technical issues that prevent the Lawyer logic from directly being applicable to proving wait-freedom. Firstly, the so-called adequacy theorem of Lawyer (that reduces establishing termination to proving a specification) is not applicable. This theorem applies to a closed program, *i.e.*, not a standalone data structure, but rather the data structure and its client. Moreover, the adequacy theorem states that when a program is proven correct in Lawyer, then all of its threads must terminate, which does not consider non-terminating clients. Hence, in this work we state and prove a separate and novel adequacy theorem that establishes wait-freedom of a standalone operation. Secondly, the wait-freedom result should apply to an arbitrary, unknown client of the operation – but, as we discussed above, it only makes sense when the client respects state encapsulation. Therefore, in our approach to verification of wait-freedom we prove once and for all that *all programs in our programming language respect state encapsulation* (if they do not use pointer arithmetic). For this purpose we construct a so-called logical relations model for our programming language. This is similar to how prior works on proving robust safety of programs [34, 12] have used logical relations models.

We use the approach outlined above to prove wait-freedom for a number of example programs written in our higher-order language, including those that highlight these higher-order programming features. In particular, we modularly verify wait-freedom of the higher order map function for lists. That is, we show that `list_map f` is wait-free whenever it is applied to any wait-free argument `f`. We also show that our approach supports practical wait-free algorithms by verifying a memory-efficient single-producer, single-consumer queue [18]. An important point about this example, and in fact many practical wait-free algorithms, is that these algorithms in fact assume fixed upper bounds on the number of concurrent operations. Verification of this class of wait-free algorithms has been largely ignored by the prior work on verification of wait-freedom. Supporting formal reasoning about such examples requires us to introduce a notion of what we call *restricted* wait-freedom, and to slightly adjust our approach.

In summary, we make the following contributions in this paper:

- A formal definition of wait-freedom for higher-order languages together with two variations needed to account for realistic examples.
- A Lawyer-based specification that internalizes the wait-freedom in the Lawyer logic.
- A novel adequacy theorem that establishes wait-freedom of an operation used by arbitrary client programs, which combines a logical relations model for safety together with Lawyer logic's approach to liveness.
- Verification of wait-freedom of a number of case studies, including those not supported by prior work.
- All results have been mechanized in the Rocq Prover (using classical axioms).

The rest of the paper is organized as follows. We formally define wait-freedom in Section 2 and also state the adequacy theorem that reduces proving wait-freedom of an operation to proving its wait-freedom specification in Lawyer. Then, in Section 3 we provide the necessary background on Iris. We then illustrate our approach in Section 4 by proving the wait-freedom specification for a simple example program. In Section 5 we continue by verifying more complicated programs which require adapting both the definition of wait-freedom and the Lawyer specification as we explained earlier. We explain the key ideas for proving our adequacy theorem in Section 6. Finally, we discuss a limitations of our solution in Section 7, compare it with related work in Section 8, and conclude in Section 9.

## 2 Overview

In this section, we formally define the notion of wait-freedom at the level of execution traces of the programming language under consideration. To avoid tedious direct reasoning about execution traces, our approach reduces proving this property to proving a specification with Lawyer program logic. We therefore briefly show the specification format of Lawyer and state the adequacy theorem for wait-freedom that establishes the aforementioned reduction.

Throughout the paper, we use the notation $m[k]$ to denote a key $k$ lookup in a map-like structure $m$. That includes $m$ being a list (and $k$ an index in it), or a partial function $m : A \xrightarrow{\text{fin}} B$ for some types $A$ and $B$ (and $k$ an element of type $A$). Moreover, we use this notation for the trace state lookup (see below). When we write $m[k]$, we implicitly assume that $k$ belongs to the domain of $m$ and thus the lookup is well-defined. Then, by $m[k \mapsto v]$ we denote the structure $m$ with the key $k$ updated to the value $v$.

Furthermore, we formally define traces and introduce a number of notations for them. For an arbitrary labeled transition system, a *trace* is a non-empty, possibly infinite (unless said otherwise) alternating sequence of states and labels of that transition system that starts and ends with a state. We write $tr[i]$ and $tr\langle i\rangle$ to denote correspondingly the $i$th state and label of a trace $tr$. Then, $\mathsf{dom}(tr)$ denotes the set of indices $i$ for which $tr[i]$ is defined, *i.e.*, natural numbers between 0 and the (potentially infinite) length of $tr$.

### 2.1 Definition of Wait-Freedom for Our Language

We consider the programs written in a higher-order ML-like language $\lambda_{\mathsf{ref}}^{\mathsf{conc}}$. An excerpt of its semantics is shown in Figure 3, for the full semantics see our Rocq development. Among the set *Expr* of $\lambda_{\mathsf{ref}}^{\mathsf{conc}}$ expressions, there are primitives for working with heap (*e.g.* $!\,e$ and $\texttt{free}\,e$) and for concurrency ($\texttt{fork}\,(e)$; the parallel composition $||$ used in the examples in Section 1 is a derived construction [2]). A less common feature of $\lambda_{\mathsf{ref}}^{\mathsf{conc}}$ – `to_int` primitive – is explained further in Section 5.2. The heap is modeled as a partial map from locations to

**Syntax**

$$
\begin{array}{llll}
x, y, f \in \mathit{Var} & & l \in \mathit{Loc} \qquad n \in \mathbb{Z} \qquad b \in \mathbb{B} & \\
& \circledcirc & ::= & + \mid - \mid * \mid = \mid < \mid \cdots \\
\mathit{Val} & v & ::= & () \mid b \mid n \mid l \mid (v, v) \mid \mathtt{inl}\, v \mid \mathtt{inr}\, v \mid \mathtt{rec}\, f\ (x) := e \\
\mathit{Expr} & e & ::= & x \mid v \mid e \circledcirc e \mid \mathtt{ref}\,(e) \mid \mathtt{if}\, e\, \mathtt{then}\, e\, \mathtt{else}\, e \mid \mathtt{fork}\,(e) \mid\ !\, e \mid e\, e \\
& & & \mid \mathtt{to\_int}\, e \mid \mathtt{free}\, e \mid \mathtt{fst}\, e \mid \mathtt{snd}\, e \\
& & & \mid \mathtt{match}\, e\, \mathtt{with}\, \mathtt{inl}\, x \Rightarrow e \mid \mathtt{inr}\, y \Rightarrow e\, \mathtt{end} \mid \ldots \\
\mathit{ECtx} & E & ::= & \bullet \mid e \circledcirc E \mid E \circledcirc v \mid (e, E) \mid (E, v) \mid e\, E \mid E\, v \mid \ldots \\
\mathit{Heap} & h & \in & \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Val} \\
\mathit{ThreadPool} & \mathcal{E} & \in & \mathsf{Thread} \xrightarrow{\text{fin}} \mathit{Expr} \\
\mathit{Config} & c & ::= & (\mathcal{E}, h)
\end{array}
$$

**Pure reductions**

$$
(\mathtt{rec}\, f\ (x) := e)\ v \overset{\text{pure}}{\rightsquigarrow} e[\mathtt{rec}\, f\ (x) := e/f][v/x]
$$

$$
\mathtt{fst}(v_1, v_2) \overset{\text{pure}}{\rightsquigarrow} v_1
$$

$$
\vdots
$$

**Per-thread reductions**

$$
(e, h) \rightsquigarrow (e', h) \quad \text{if } e \overset{\text{pure}}{\rightsquigarrow} e'
$$

$$
(\mathtt{ref}\,(v), h) \rightsquigarrow (\ell, h[\ell \mapsto v]) \quad \text{if } \ell \notin \mathsf{dom}(h)
$$

$$
\vdots
$$

**Configuration reductions**

$$
\frac{(e, h) \rightsquigarrow (e', h')}{(\mathcal{E}[\tau \mapsto \mathit{fill}\, E\, e], h) \xrightarrow{\tau} (\mathcal{E}[\tau \mapsto \mathit{fill}\, E\, e'], h')}
$$

$$
\frac{\tau' \notin \mathsf{dom}(\mathcal{E}) \cup \{\tau\} \quad \mathcal{E}' \triangleq \mathcal{E}[\tau \mapsto \mathit{fill}\, E\, ()][\tau' \mapsto e]}{(\mathcal{E}[\tau \mapsto \mathit{fill}\, E\, (\mathtt{fork}\,(e))], h) \xrightarrow{\tau} (\mathcal{E}', h)}
$$

**Figure 3** An excerpt of $\lambda_{\mathsf{ref}}^{\mathsf{conc}}$ semantics.

language values *Val* (which might include closures). In turn, the thread pool is modeled as a partial map from thread identifiers Thread to expressions; the per-thread reductions are naturally lifted to configurations *Config* (thread pools together with heaps). The structure of evaluation contexts *ECtx* establishes right-to-left call-by-value evaluation order. The $\mathit{fill}\,(E : \mathit{ECtx})\,(e : \mathit{Expr})$ operator fills the hole $\bullet$ in the context $E$ with the expression $e$; we avoid the more common $E[e]$ notation, as in this paper it is used for lookups (see above).

Note that $\lambda_{\mathsf{ref}}^{\mathsf{conc}}$ forms a labeled transition system where states are configurations, labels are thread identifiers, and transitions are configuration reductions. In particular, for an execution trace *etr* of $\lambda_{\mathsf{ref}}^{\mathsf{conc}}$ and any index $i \in \mathsf{dom}\, tr$, $tr[i]$ is the configuration before the $i$th step, and $etr\langle i \rangle$ is the identifier of the thread taking the $i$th step.

We define wait-freedom as a property of an *operation* $\mathsf{op} : \mathit{Val}$ – an arbitrary one-argument function. Intuitively, op is wait-free if any "call" to op is followed by a "return" later in the execution. A "call" in $\lambda_{\mathsf{ref}}^{\mathsf{conc}}$ is simply an application $\mathsf{op}\, a$ for some $(a : \mathit{Val})$. To identify a call in a thread pool $\mathcal{E}$, we need to provide a thread identifier $\tau : \mathsf{Thread}$ and an evaluation context $E : \mathit{ECtx}$. Then, a return in our approach is simply a value, similarly identified with a thread identifier and an evaluation context.

▶ **Definition 1** (Call and return in a thread pool).

$$
\mathit{Call}(\mathcal{E}, \tau, E, \mathsf{op}) \triangleq \exists a \in \mathit{Val}.\, \mathcal{E}[\tau] = \mathit{fill}\, E\, (\mathsf{op}\, a) \qquad \mathit{Ret}(\mathcal{E}, \tau, E) \triangleq \exists r \in \mathit{Val}.\, \mathcal{E}[\tau] = \mathit{fill}\, E\, r
$$

Requiring that every call to op is followed by return only makes sense when the calling thread is kept being scheduled until the return. This is captured by the following definition.

▶ **Definition 2** (Eventual return of calls). *For an operation* $\mathsf{op} : \mathit{Val}$*, execution trace* $\mathit{etr}$ *and thread identifier* $\tau$*, we say that all* $\tau$*'s calls to* $\mathsf{op}$ *in* $\mathit{etr}$ *eventually return iff*

$$\mathit{alwaysReturnsInTrace}(\mathsf{op}, \mathit{etr}, \tau) \triangleq \forall (E : \mathit{ECtx}), i.$$
$$\mathit{Call}(\mathit{etr}[i].1, \tau, E, \mathsf{op}) \Rightarrow \mathit{schedUntilRet}(\mathit{etr}, \tau, E, i) \Rightarrow \exists j \geq i.\ \mathit{Ret}(\mathit{etr}[j].1, \tau, E)$$
$$\text{where } \mathit{schedUntilRet}(\mathit{etr}, \tau, E, i) \triangleq \forall j \geq i.\ j \in \mathsf{dom}\ \mathit{etr} \Rightarrow$$
$$\neg(\exists k \in [i.\,.j].\ \mathit{Ret}(\mathit{etr}[k].1, \tau, E)) \Rightarrow \exists p \geq j.\ \mathit{etr}\langle p \rangle = \tau$$

Above, $\mathit{schedUntilRet}(\mathit{etr}, \tau, E, i)$ says that, if the thread $\tau$ has called (in terms of Definition 1) $\mathsf{op}$ at index $i$ and has not returned from it yet, then $\tau$ is guaranteed to be eventually scheduled. Note that each call requires a separate return, matched by an evaluation context. Indeed, in $\lambda^{\mathsf{conc}}_{\mathsf{ref}}$, an evaluation context is preserved by configuration reductions until the expression under that context reduces to a value. Therefore, the contexts of, *e.g.*, nested calls will be different (with every subsequent call occurring in a "deeper" context).

Finally, we consider wait-freedom in executions with the following starting configurations:

- The program of every thread must be a substitution of the wait-free operation's code into some expression with a free variable $x$. Moreover, this expression must not contain hard-coded locations and pointer arithmetic (denoted by *noLocs*): the example from Figure 2 shows why it is necessary to restrict random memory access in clients.
- The data structure accessed by the operation must be initialized. What exactly "initialization" means depends on the operation (see an example below), therefore our definition of wait-freedom is parameterized with the set of initialized configurations.

As a result, we define wait-freedom of an operation as requiring that operation to always return in any thread of any execution that starts in an appropriately initialized configuration.

▶ **Definition 3** (Wait-freedom). *Given a set of initialized configurations* $C \subseteq \mathit{Config}$*, we define wait-freedom of an operation* $\mathsf{op}$ *as*

$$\mathit{waitFree}_C(\mathsf{op}) \triangleq \forall \mathit{etr}.\ \bigwedge_{\tau \mapsto e \in \mathit{etr}[0].1} (\exists e'.\ e = e'[\mathsf{op}/x] \wedge \mathit{noLocs}(e')) \Rightarrow$$
$$\mathit{etr}[0] \in C \Rightarrow \forall \tau.\ \mathit{alwaysReturnsInTrace}(\mathsf{op}, \mathit{etr}, \tau)$$

For example, in Section 4 we show that a FAA-based `incr` operation of counter (Figure 1c) is wait-free if the counter location $l$ is initialized with an integer. In terms of Definition 3, it is denoted as $\mathit{waitFree}_{\mathit{hasInt}(l)}(\texttt{incr}\ l)$, where $\mathit{hasInt}(l) \triangleq \{c \mid c.2[l] \in \mathbb{Z}\}$.

Note that, while Definition 3, inspired by the classical definition by Herlihy [15], captures the behavior of simple wait-free operations, more complicated classes of algorithms only satisfy weaker forms of that property. We will see examples of it in Section 5.

## 2.2 Proving Wait-Freedom with Lawyer Logic

Definition 3 captures wait-freedom at the level of execution traces, which are notoriously hard to reason about directly. To avoid that, our approach reduces establishing wait-freedom of a given operation to verifying this operation with Lawyer [31] – a concurrent separation logic built on top of Iris [22] and Trillium [35] frameworks. We explain the details of Lawyer specifications and logic in the subsequent sections; here, we present the high-level form of the specification that a wait-free operation must satisfy.

▶ **Definition 4** (Lawyer specification of wait-freedom). *For a set of initialized configurations* $C \subseteq \mathit{Config}$*, we define the Lawyer specification of* $\mathsf{op}$*'s wait-freedom as*

$$\mathsf{WaitFreeSpec}_C(\mathsf{op}) \triangleq \forall c \in C.\ \mathsf{heap\_repr}(c) \Rrightarrow \mathsf{NoInfExec}(\mathsf{op}) \wedge \mathsf{PresInv}(\mathsf{op})$$

$$P, Q ::= \mathsf{True} \mid \mathsf{False} \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \mid \forall x.\, P \mid \exists x.\, P \mid P * Q \mid$$
$$\{P\}\,(e : \mathit{Expr})\,\{v.Q\} \mid (\ell : \mathit{Loc}) \mapsto (v : \mathit{Val}) \mid \boxed{P} \mid \mathsf{toks}_N(k) \mid P \Rrightarrow Q \mid \ldots$$

**Figure 4** Propositions of Iris logic. Here, *Expr*, *Loc* and *Val* refer to those from Figure 3.

The above specification can be read as follows. For any initialized configuration $c$, we assume the ownership of $c$'s entire heap, represented as an Iris proposition $\mathsf{heap\_repr}(c)$. Given that (reading $\Rrightarrow$ as an implication), we prove two specifications for the operation $\mathtt{op}$. First, $\mathsf{NoInfExec}(\mathtt{op})$ states that for any argument $a : \mathit{Val}$, execution of $\mathtt{op}\,a$ cannot be infinite. Second, $\mathsf{PresInv}(\mathtt{op})$ states that for any $a$, execution of $\mathtt{op}\,a$ preserves the internal invariants that $\mathtt{op}$ relies on (*e.g.* that the location $l$ of counter in Figure 1c always stores a number).

Our adequacy theorem states that the specification above indeed implies wait-freedom.

▶ **Theorem 5** (Adequacy). *If* $\mathsf{WaitFreeSpec}_C(\mathtt{op})$ *is provable in Lawyer, then* $\mathit{waitFree}_C(\mathtt{op})$ *holds in the meta-logic.*

As we note in Section 2.1, the notion of wait-freedom for more complicated classes of operations is more elaborate. For these, we use slightly different versions of Definition 4 and Theorem 5 described in Section 5.

## 3 Background: Iris Logic

The Lawyer logic [31] that we use to verify wait-freedom is built on top of Trillium framework [35], which is in turn an extension of the Iris framework [22]. In this section, we provide the general background on Iris by proving two specifications (not relevant to wait-freedom) for the counter increment operation `incr` from Figure 1c. Later in Section 4, we actually verify the wait-freedom specification for `incr` with Lawyer logic, concentrating on how it differs from Iris. We only address Trillium framework specifically in Theorem 10.

Strictly speaking, Iris is a *framework* that only provides the minimal base logic and allows to define custom program logics on top of it. We consider the specific instantiation of Iris (inspired by one used in Iris Lecture Notes [2]) shown in Figure 4, the constructs of which we explain as we go. In this section, by "Iris" we mean that specific instantiation. Similarly, there are many possible specifications for `incr` even within that Iris instantiation; we consider those that illustrate the concepts used in this paper.

The simplest application of Iris is using it as a *sequential Hoare logic*. Such logics reduce verifying properties of sequential programs to proving *Hoare triples* for them. Such a triple is written as $\{P\}\,e\,\{v.\,Q\}$ and intuitively means that if the precondition $P$ holds when an execution of program $e$ starts, then the execution does not get stuck, and that whenever $e$ reduces to some value $v$, the postcondition $Q(v)$ holds. The program state described by pre- and post-conditions can include, among others, the *points-to* propositions. Such a proposition is written as $\ell \mapsto v$, meaning that the heap location $\ell$ currently has value $v$; moreover, it asserts exclusive ownership of $\ell$ – the notion we address later. Finally, Iris includes the standard primitives of first-order logic such as propositional connectives and quantifiers.

With the intuition above, we can specify the program in Figure 1c as follows:

$$\forall \ell : \mathit{Loc}, n : \mathbb{Z}.\, \{\ell \mapsto n\}\,\mathtt{incr}\,\ell\,()\,\{m.\, m = n \wedge \ell \mapsto (n+1)\} \qquad \text{(incr-spec-seq)}$$

This specification states that regardless of the location and the current value of counter, the increment operation executes without getting stuck, that its return value equals to the current counter value, and that upon return this value is actually incremented. Indeed, the adequacy theorem of Iris [20] allows to establish this property.

$$\text{IRIS-BETA}\ \frac{\{P\}\, e[v/x]\, \{Q\}}{\{P\}\, (\lambda x.e)v\, \{Q\}} \qquad \text{IRIS-FAA}\ \{\ell \mapsto n\}\, \texttt{FAA}(\ell, k)\, \{m.\, m = n \land \ell \mapsto (n+k)\}$$

$$\text{IRIS-HT-FRAME}\ \frac{\{P\}\, e\, \{Q\}}{\{P * R\}\, e\, \{Q * R\}}$$

$$\text{IRIS-HT-CSQ}\ \frac{\{P'\}\, e\, \{Q'\} \quad P \Rightarrow P' \quad \forall v.\, Q'(v) \Rightarrow Q(v)}{\{P\}\, e\, \{Q\}} \qquad \text{IRIS-INV-ACC}\ \frac{\{P * Q\}\, e\, \{P * R\} \quad e \text{ is atomic}}{\left\{\boxed{P} * Q\right\}\, e\, \{R\}}$$

**Figure 5** Selected rules of Iris program logic.

Proving simple specifications such as incr-spec-seq amounts to applying primitive proof rules on every execution step. Figure 5 shows a subset of Iris proof rules required to verify the specifications in this section. In particular, as execution of $\texttt{incr}\,\ell\,()$ consists of two beta-reductions and one FAA step, we need the rules IRIS-BETA and IRIS-FAA. The former states that taking a beta-reduction does not change the program state, so the reduct can be verified with the same pre- and post-condition. The latter states that fetch-and-add updates the value stored under the corresponding location and returns the previously stored value.

To prove incr-spec-seq, we first introduce the universally quantified $\ell$ and $n$. Then, we apply IRIS-BETA twice to reduce `incr` and `incr_waitfree`. Afterwards, we apply IRIS-FAA to prove $\{\ell \mapsto n\}\, \texttt{FAA}(\ell, 1)\, \{m.\, m = n \land \ell \mapsto (n+1)\}$ and finish the proof.

However, the specification above and the fragment of Iris logic described so far have a number of disadvantages. First, the pre- and postconditions of Hoare triples describe the entire program state, whereas the rules such as IRIS-FAA are intended to operate only with a part of it. Then, the specification incr-spec-seq is *sequential*: it assumes that the thread executing $\texttt{incr}\,\ell$ always knows the exact value under $\ell$. This is not a realistic case for *concurrent* libraries: other threads can interfere by running $\texttt{incr}\,\ell$ on their own.

Below, we illustrate the use of Iris as a *concurrent separation logic* by proving a more elaborate specification for `incr`. There are many possible concurrent specifications for this operation; Iris Lecture Notes [2] provide a number of them. We choose another, *token-based* specification because the notion of "tokens" is used a few times throughout the paper: it resembles the Lawyer's notion of fuel (Section 4), and is used to specify restricted wait-freedom (Section 5.2). As said above, in the concurrent setting it is impossible to always hold the exact knowledge of the current counter value. However, by limiting the number of possible increments, one can approximate the current counter value when reading it. Namely, when instantiating the counter, the user chooses its maximal possible value $N$. Upon the instantiation, the user receives $N$ non-replenishable *tokens*. A token is spent on every counter increment. Unused tokens can be presented upon reading the counter; the more tokens are presented, the smaller the counter value must be. Formally, we can specify the counter operations as follows (where $\texttt{mk\_cnt}\,() \triangleq \texttt{ref}\,0$ and $\texttt{read}\,\ell \triangleq !\ell$):

$$\forall N : \mathbb{N}. \quad \{\mathsf{True}\}\, \texttt{mk\_cnt}\,()\, \left\{\ell.\, \boxed{\mathsf{cntInvUpto}(\ell, N)} * \mathsf{toks}_N(N)\right\}$$

$$\forall \ell, N. \quad \left\{\boxed{\mathsf{cntInvUpto}(\ell, N)} * \mathsf{toks}_N(1)\right\}\, \texttt{incr}\,\ell\,()\, \{m.\, 0 \le m\}$$

$$\forall \ell, N, k. \quad \left\{\boxed{\mathsf{cntInvUpto}(\ell, N)} * \mathsf{toks}_N(k)\right\}\, \texttt{read}\,\ell\, \{m.\, 0 \le m \le (N-k) * \mathsf{toks}_N(k)\}$$

This specification features all the main concepts of Iris used to verify concurrent programs. First, it shows that Iris is a *separation logic* that treats propositions as *disjoint resources*. Resources represent both the state of the program (*e.g.* points-to propositions) and the auxiliary information (*e.g.* the number of tokens). The *separating conjunction* $P * Q$ asserts ownership of a resource that can be split into two disjoint parts satisfying $P$ and $Q$ correspondingly. The notion of being disjoint depends on particular resource kind; for points-to, $l_1 \mapsto v_1 * l_2 \mapsto v_2$ implies $l_1 \neq l_2$, *i.e.*, ownership of a memory location is *exclusive*.

Then, $\mathsf{toks}_N(k)$ is an example of a *ghost resource*, *i.e.* one that is not directly tied to the program state. The Iris user can choose a number of ghost resources to support the verification process. In Figure 4, we assume that Iris instantiation contains resources for tracking tokens, where a proposition $\mathsf{toks}_N(k)$ asserts ownership of $k$ out of $N$ tokens and satisfies the following laws:

$$\mathsf{toks}_N(k) * \mathsf{toks}_N(m) \dashv\vdash \mathsf{toks}_N(k+m) \qquad \text{(tokens-sep)}$$

$$\mathsf{toks}_N(k) \vdash k \leq N \qquad \text{(tokens-le)}$$

Finally, the specifications above feature an *invariant* $\boxed{\mathsf{cntInvUpto}(\ell, N)}$. Once established, an invariant proposition $\boxed{P}$ means that $P$ will hold throughout program's execution. More specifically, it is guaranteed to hold between the execution steps, so every thread can rely on $P$ to justify its own step, but must reestablish $P$ after taking that step. Importantly, invariants (but not their content in general) are duplicable. That is, $\boxed{P} \vdash \boxed{P} * \boxed{P}$; with that, exclusive resources such as points-to can be placed in the invariant and accessed by every thread through its own copy of invariant. For the specification above, we define the counter invariant as follows, capturing the intuition of spending a token for every increment:

$$\mathsf{cntInvUpto}(\ell, N) \triangleq \exists k \geq 0.\, \ell \mapsto k * \mathsf{toks}_N(k)$$

We omit the verification of `mk_cnt` and `read` as it follows standard principles of Iris safety proofs [2]. Below, we show how the verification of `incr` changes with the new specification. We verify the two beta-reductions as before by applying IRIS-BETA twice and end up with the following triple to prove:

$$\left\{\boxed{\mathsf{cntInvUpto}(\ell, N)} * \mathsf{toks}_N(1)\right\} \texttt{FAA}(\ell, 1)\, \{m.\, 0 \leq m\}$$

To proceed, we need the points-to resource in the invariant. For this, we use the invariant access rule IRIS-INV-ACC that allows to open the invariant (*i.e.* obtain the underlying resource) for a single execution step and requires to close it back after that step[1]. Applying this rule to the triple above and unfolding our invariant, we are left with proving

$$\{(\exists k \geq 0.\, \ell \mapsto k * \mathsf{toks}_N(k)) * \mathsf{toks}_N(1)\}\, \texttt{FAA}(\ell, 1)\, \{m.\, (\exists k \geq 0.\, \ell \mapsto k * \mathsf{toks}_N(k)) * 0 \leq m\}$$

Hoare triples enjoy the rule of consequence IRIS-HT-CSQ. Using it along with TOKENS-SEP and standard rules for separation logic, we reduce the triple above to the following, for some $k \geq 0$:

$$\{\ell \mapsto k * \mathsf{toks}_N(k+1)\}\, \texttt{FAA}(\ell, 1)\, \{m.\, m = k * \ell \mapsto (k+1) * \mathsf{toks}_N(k+1)\}$$

[1] To prevent unsoundly opening the invariant twice in the same step, Iris features *masks* of Hoare triples and *namespaces* of invariants. We omit them for simplicity. Moreover, in general the resource under invariant is obtained under the *later modality*; for this particular invariant it can be ignored.

To verify the FAA step, we first apply the frame rule IRIS-HT-FRAME that strips the irrelevant resources from both the pre- and post-condition. We close the proof by applying IRIS-FAA to verify the remaining triple $\{\ell \mapsto k\}\,\texttt{FAA}(\ell, 1)\,\{m.\, m = k * \ell \mapsto (k+1)\}$.

In the subsequent sections, specifications will be verified similarly to what we did above. There will be slight differences in the Hoare triples format and the exact form of proof rules, and we will use other ghost resources, but the overall idea will be the same: establish the invariant and then verify every execution step with the corresponding proof rule.

## 4 Verifying Wait-Freedom in Lawyer Logic

In this section, we show how an operation is verified against the wait-freedom specification of Definition 4. As an example, we consider $\texttt{incr}\,l$ from Figure 1c. Specifically, we verify $\mathsf{WaitFreeSpec}_{\mathit{hasInt}(l)}(\texttt{incr}\,l)$ in the Lawyer logic for any location $l$ (see Section 2 for the definition of $\mathit{hasInt}(l)$). Doing so allows us to use Theorem 5 to conclude $\mathit{waitFree}_{\mathit{hasInt}(l)}(\texttt{incr}\,l)$ holds at the meta-level. Most of the verification process amounts to proving the Hoare triples similarly to how it was done in Section 3, although the details of program logics are different.

Consider some counter location $l$. By definition of $\mathsf{WaitFreeSpec}$, we need to prove

$$\forall c \in \mathit{hasInt}(l).\, \mathsf{heap\_repr}(c) \Rrightarrow \mathsf{NoInfExec}(\texttt{incr}\,l) \wedge \mathsf{PresInv}(\texttt{incr}\,l)$$

To do so, we take an arbitrary configuration $c : \mathit{Config}$ for which we have $c.2[l] = k$ for some $k \in \mathbb{Z}$, and further assume ownership of each heap location of $c$ with separating conjunction over all available points-to resources:

$$\mathsf{heap\_repr}(c) \triangleq \mathop{\Large *}_{\{(\ell,v)\ \mid\ c.2[\ell]=v\}} \ell \mapsto v$$

With the ownership of the entire heap, we will prove the specifications $\mathsf{NoInfExec}(\texttt{incr}\,l)$ and $\mathsf{PresInv}(\texttt{incr}\,l)$. Proving both relies on the fact that $l$ always stores some number. For this, we use a counter module invariant (which is even simpler than one used in Section 3):

$$\mathsf{cntInv} \triangleq \boxed{\exists k \in \mathbb{Z}. l \mapsto k}$$

Note that the definition of $\mathsf{WaitFreeSpec}$ features a so-called *viewshift*, written $\Rrightarrow$. Viewshifts are similar to logical implication, but additionally allow to access and establish the invariants so as to establish the consequent. Thus, given ownership of the entire heap, and the fact that $l$ currently points to some integer, we establish $\mathsf{cntInv}$ by providing $l \mapsto k$ (ignoring the rest of the heap). From that point on we assume that $\mathsf{cntInv}$ holds while proving the specifications (by adding it to preconditions of Hoare triples under consideration).

The first specification we need to prove, $\mathsf{NoInfExec}(\texttt{incr}\,l)$, informally captures that execution of our operation applied to any argument cannot run forever. Formally, it is defined as a dependent pair of a natural number and a specification that mentions it[2][3]:

$$\mathsf{NoInfExec}(\mathsf{op}) \triangleq \{\mathbf{F} : \mathbb{N} \mid \forall \tau : \mathsf{Thread}, \pi : \mathsf{Phase}, a : \mathit{Val}.\, \{\mathbf{F} \cdot \mathsf{bar}_\circ\, \pi * \mathsf{ph}_\circ\, \tau\, \pi\}\ \mathsf{op}\ a\ \{\mathsf{ph}_\circ\, \tau\, \pi\}^\tau\}$$

The number $\mathbf{F}$ is (an over-approximation of) the upper bound on the execution time of $\mathsf{op}$. The actual specification for $\mathsf{op}$ is defined as a *Lawyer Hoare triple*. Lawyer [31] is an instantiation of the Trillium framework, which is in turn an extension of Iris that establishes

[2] The reason why $\mathbf{F}$ cannot be existentially quantified inside the triple is related to the issues of step-indexing which are described in detail by Spies *et al.* [33].

[3] We omit the return value binder $v$, because it is not used in the postcondition.

*refinement* of a program to a transition system. In particular, compared to the triples of Iris introduced in Section 3, the Hoare triples in Lawyer additionally enforce that every single step of the program is matched by a transition in so-called Obligations Model – a transition system that Lawyer uses to represent terminating programs. Proving a Lawyer Hoare triple for a program implies that any execution trace of that program refines some trace of the Obligations Model. As all traces of the Obligation Model are finite, this refinement implies termination of verified programs. Note that the Lawyer triples are additionally annotated with the thread identifier, because the thread identity is important for proving refinement.

The precondition and postcondition of the triple in $\mathsf{NoInfExec}(\mathsf{op})$ consist of the ghost resources of Lawyer logic. The precondition can be read as an assignment of $\mathbf{F}$ barrels of *fuel* to the thread $\tau$, with an indirection via the *phase* $\pi$ that marks the barrels and is assigned to $\tau$. The fuel is represented by the $\mathbf{F}\cdot\mathsf{bar}_\circ\pi$ resource (a separating conjunction of $\mathbf{F}$ many individual barrels $\mathsf{bar}_\circ\pi$). The phase assignment is written in the Lawyer logic as $\mathsf{ph}_\circ\tau\,\pi$. The fuel resource represents the number of remaining steps the program is allowed to take: as we will see later, verifying every single execution step consumes one barrel of fuel, similar to how every counter increment consumed a token in Section 3. We remark that in Lawyer [31], both the fuel and the phase of resources have one extra parameter each. Technically, we assign these parameters the values that are largely irrelevant for this presentation; instead of showing these values, we just mark both resources with a $\circ$ subscript. In the appendix, we argue why this particular specification pattern of $\mathsf{NoInfExec}(\mathsf{op})$, as well as certain restrictions on the operation's invariants, captures the nature of wait-freedom in the Lawyer logic.

To prove $\mathsf{NoInfExec}(\texttt{incr}\,l)$, we take $\mathbf{F}$ to be 3 and assume the invariant established earlier. Then, we prove, for any $\tau$, $\pi$, and $a$:

$$\{\mathsf{cntInv} * 3\cdot\mathsf{bar}_\circ\pi * \mathsf{ph}_\circ\tau\,\pi\}\ \texttt{incr}\,l\,a\ \{\mathsf{ph}_\circ\tau\,\pi\}^\tau$$

The triple above essentially states that we can spend up to 3 barrels of fuel to complete the execution of $\mathsf{op}\,a$. To prove it, we use the rules[4] of the Lawyer logic. Compared to the proof rules presented in Section 3, the main difference is that the rules of Lawyer feature Lawyer-specific resources besides the Iris ones. This is because Lawyer requires to match every step of execution with a transition in Obligations Model, which in turn requires resources such as fuel. Specifically, the Lawyer rules for beta-reductions and FAA are the following:

LAWYER-BETA-FUEL
$$\{\mathsf{ph}_\circ\tau\,\pi * P\}\ e[v/x]\ \{\mathsf{ph}_\circ\tau\,\pi * Q\}^\tau \vdash \{\mathsf{bar}_\circ\pi * \mathsf{ph}_\circ\tau\,\pi * P\}\ (\lambda x.\ e)\,v\ \{\mathsf{ph}_\circ\tau\,\pi * Q\}^\tau$$

LAWYER-FAA-FUEL
$$\{\mathsf{bar}_\circ\pi * \mathsf{ph}_\circ\tau\,\pi * \ell \mapsto n\}\ \texttt{FAA}(\ell, k)\ \{m.\, m = n * \ell \mapsto (n+k) * \mathsf{ph}_\circ\tau\,\pi\}^\tau$$

After applying LAWYER-BETA-FUEL twice, we are left to prove the following:

$$\{\mathsf{cntInv} * \mathsf{bar}_\circ\pi * \mathsf{ph}_\circ\tau\,\pi\}\ \texttt{FAA}(l, 1)\ \{\mathsf{ph}_\circ\tau\,\pi\}^\tau$$

Proving it requires opening the invariant, for which Lawyer inherits the IRIS-INV-ACC rule unchanged. Applying it, along with IRIS-HT-CSQ and LAWYER-FAA-FUEL, finishes the proof.

The next specification that we need to prove, *i.e.*, $\mathsf{PresInv}(\texttt{incr}\,l)$, is defined as a yet another Hoare triple as follows:

$$\mathsf{PresInv}(\mathsf{op}) \triangleq \forall\tau : \mathsf{Thread}, a : \mathit{Val}.\ \{\mathsf{RS}_{\mathbf{V}}(a)\}^{\lightning}\ \mathsf{op}\ a\ \{v.\ \mathsf{RS}_{\mathbf{V}}(v)\}^\tau$$

[4] The rules that we present below are derived from the more general rules of Lawyer logic.

As its name suggests, this triple guarantees that the execution of `op` preserves the established invariants. In fact, the same holds for Hoare triples of Iris and Lawyer; the crucial difference is that this triple allows the execution to get stuck (denoted by $\lightning$ superscript). This triple does not establish refinement to any model, but for technical reasons it is annotated with the thread identifier like a Lawyer triple[5].

As we will explain in Section 6.2, such Hoare triples ensure that `op` can be used by an arbitrary client program, and that it would not violate the internal invariants of `op`. Technically, a client program *uses* `op` by providing it a *safe* argument, captured by the predicate $\mathsf{RS}_{\mathbf{V}}(a)$, and expects `op` to return a safe value back. We will explain what it means to be safe and motivate why it is necessary to restrict only to safe values in Section 6.2.

Since $\mathsf{PresInv}(\texttt{incr}\,l)$ does not establish refinement to a model, its proof follows the process described in Section 3. We ignore the $\mathsf{RS}_{\mathbf{V}}(a)$ resource, because `incr` ignores the input argument $a$. The rules used to verify beta-reductions and FAA are exactly the same as in Section 3 (except for the different type of triple). Similarly to the proof above, we use $\mathsf{cntInv}$ to verify the FAA step. The main difference there is that the return value $v$ is not ignored – it needs to be shown to be safe, *i.e.* $\mathsf{RS}_{\mathbf{V}}(v)$. However, we know that FAA returns an integer, and, as we will see in Section 6.2, all integers are safe, *i.e.*, $\forall n : \mathbb{Z}.\ \vdash \mathsf{RS}_{\mathbf{V}}(n)$.

# 5 Case Studies

As we discussed in the Introduction, while the simple definition of wait-freedom that we used in Sections 2 and 4 works for many simple programs, for more involved programs, this definition needs to be adjusted. Specifically, Definition 3 follows the classical definition of wait-freedom [15]: it requires an operation to progress safely (*i.e.*, without getting stuck) regardless of its input arguments and the number of concurrent operations. However, in practice each of these conditions can affect the safe progress.

In this section, we consider two illustrative examples that motivate two independent extensions to our wait-freedom verification approach. First, we verify a (higher-order) function that gets stuck on a "wrong" argument. Second, we verify a data structure that restricts the number of operations that can run concurrently at a given time. The latter is especially important, as many realistic wait-free data structures [13, 25, 36] rely on this restriction. We note that these changes account for different types of progress properties, not for each individual example; we believe that the case studies exhibiting the same property can be verified with the same specification and the adequacy theorem.

Throughout this section we will use the helper definitions `None` $\triangleq$ `inl` () and `Some` $v \triangleq$ `inr` $v$ and similarly define a corresponding notation for in pattern matching where we simply write `match e with None => e1 | Some v => e2 end`.

## 5.1 Modular Verification of Higher-Order, Possibly-Stuck Operations

Consider the higher-order `list_map` function in Figure 6a. It applies the given function `f` to all elements of the list $l$. In our language, lists are values satisfying the recursive predicate $\mathsf{isList}$ defined in Figure 6b.

[5] Technically, this Hoare triple is defined by directly instantiating the Trillium's program logic with a trivial, always looping transition system.

```
let rec list_map f l =
  match l with
  | None => None
  | Some p => Some (f (fst p), list_map f (snd p))
  end
```

**(a)** Implementation.

$$\mathsf{isList}(v : \mathit{Val}) \triangleq v = \mathbf{None} \vee \exists h, v'.\, v = \mathbf{Some}(h, v') \wedge \mathsf{isList}(v')$$

**(b)** Representation of lists in $\lambda_{\mathsf{ref}}^{\mathsf{conc}}$.

**Figure 6** Higher-order list mapping function.

Note that if the input function `f` does not run forever for any argument, the partially applied `list_map f` does not run forever for any argument either. However, it might get stuck even if `f` never does: if its second argument $l$ does not satisfy isList, then `list_map` will get stuck, *e.g.*, on taking projections. Thus, the progress of calls to `list_map f` is captured by the following variant of Definition 2, with the changes highlighted:

▶ **Definition 6** (Eventual return or stuckness of calls to op)**.** *For an operation* op : *Val, execution trace etr and thread identifier* $\tau$*, we say that all* $\tau$*'s calls to* op *in etr eventually return or get stuck, written* $\mathit{alwaysReturnsInTrace}^{\lightning}(\mathtt{op}, \mathit{etr}, \tau)$*, iff*

$$\begin{aligned}&\mathit{alwaysReturnsInTrace}^{\lightning}(\mathtt{op}, \mathit{etr}, \tau) \triangleq\\ &\quad \forall \mathcal{E}, i.\ \ \mathit{Call}(\mathit{etr}[i].1, \tau, E, \mathtt{op}) \wedge \mathit{schedUntilRet}(\mathit{etr}, \tau, E, i) \Rightarrow\\ &\quad \exists j \geq i.\ \mathit{Ret}(\mathit{etr}[j].1, \tau, E) \vee \mathit{stuck}(\mathit{etr}[j], \tau)\\ &\mathit{where\ stuckness\ is\ defined\ as}\ \mathit{stuck}(c, \tau) \triangleq \neg\exists c'.\, c \xrightarrow{\tau} c'\end{aligned}$$

Similarly, we define possibly-stuck wait-freedom $\mathit{waitFree}^{\lightning}$ as a variant of Definition 3 that uses $\mathit{alwaysReturnsInTrace}^{\lightning}$ instead of $\mathit{alwaysReturnsInTrace}$.

Our approach to proving wait-freedom supports verifying possibly-stuck wait-freedom with minimal changes compared to the ordinary wait-freedom discussed in earlier sections. That is, we prove a variant of Theorem 5 that establishes possibly-stuck wait-freedom of an operation, given that it satisfies the Lawyer specification $\mathsf{WaitFreeSpec}^{\lightning}$, which, similarly to definitions above, is a weakening of WaitFreeSpec that allows the operation to get stuck.

Now, we can apply the approach outlined in Sections 2 and 4 to prove that `list_map f` is possibly-stuck wait-free. Crucially, we do so modularly by verifying both `list_map` and `f` independently of one another. Formally, we prove the following higher-order possibly-stuck wait-freedom specification for `list_map`:

$$\forall \mathtt{f}, C.\, \mathsf{WaitFreeSpec}_C^{\lightning}(\mathtt{f}) \Rightarrow \mathsf{WaitFreeSpec}_C^{\lightning}(\mathtt{list\_map\ f}) \qquad \text{(list-map-HO-spec)}$$

As in Section 4, we prove the two specifications for `list_map f`. For each of them, we make use of the corresponding specification for `f`. The logical inference rules used in the process are the same used in Section 4, with the addition of rules to verify stuck computations, *e.g.*, the following Hoare triple for projecting of the unit value: $\{\mathsf{True}\}^{\lightning}\ \mathbf{fst}\ ()\ \{P\}^{\tau}$.

Given the (list-map-HO-spec) spec above, together with the fact that ordinary Hoare triples imply possibly-stuck Hoare triples, we can reuse the already proven (possibly-stuck) wait-freedom specifications to derive possibly-stuck wait-freedom of `list_map` partially applied to specific function argument. For example, by weakening the Hoare triple that we proved in Section 4, we obtain a proof of $\mathsf{WaitFreeSpec}^{\lightning}_{\mathit{hasInt}(l)}(\mathtt{list\_map\ (incr}\ l\mathtt{)})$ for any location $l$, thus establishing $\mathit{waitFree}^{\lightning}_{\mathit{hasInt}(l)}(\mathtt{list\_map\ (incr}\ l\mathtt{)})$.

```
(* Node = {val: Value; next: pointer to Node};   OldHeadVal: Value *)
(* Head, Tail, BeingRead, FreeLater : pointer to Node *)
```

```
(* methods used by "enqueuer" thread: *)

let read_head_e () =
  let cur_head = !Head in
  if (cur_head == !Tail) then None
  else
    BeingRead := cur_head;
    let v =
      if (cur_head == !Head) then
        cur_head.val
      else
        !OldHeadVal
    in Some v

let enqueue v =
  let dummy = Node {val: 0, next: null} in
  let new_tail = ref dummy in
  let cur_tail = !Tail in
  cur_tail.val := v;
  cur_tail.next := new_tail;
  Tail := new_tail
```

```
(* methods used by "dequeuer" thread: *)

let deque () =
  let cur_head = !Head in
  if (cur_head == !Tail) then None
  else
    let v = cur_head.val in
    OldHeadVal := v;
    Head := cur_head.next;
    let to_free =
      if (cur_head == !BeingRead) then
        let old_read = !FreeLater in
        FreeLater := cur_head;
        old_read
      else cur_head
    in free to_free; Some v

let read_head_d () =
  let cur_head = !Head in
  if (cur_head == !Tail) then None
  else Some cur_head.val
```

**(a)** Original implementation by Jayanti and Petrovic [18] expressed in $\lambda_{\text{ref}}^{\text{conc}}$.

```
let enqueuer v =
  if (v == ()) then
    Some (read_head_e ())
  else
    match (to_int v) with
    | Some n => Some (enqueue n)
    | None => None
    end
```

```
let dequeuer v =
  if (v == true) then
    Some (read_head_d ())
  else if (v == false) then
    Some (deque ())
  else
    None
```

**(b)** Wrappers to be verified.

**Figure 7** Single-enqueuer, single-dequeuer queue.

Finally, we note that verifying `list_map` requires one more generalization to the specification (unrelated to possible stuckness). The amount of fuel consumed by `list_map f l` depends on the length of the input list `l`; to account for that, we replace the fuel amount $\mathbf{F}$ in NoInfExec with a *fuel function* of the type $\mathit{Val} \to \mathbb{N}$. In fact, the fuel amount used in all other examples is simply defined as a constant fuel function. We refer the reader to our Rocq mechanization for details.

## 5.2 Restricted Wait-Freedom

Many practical implementations of wait-free algorithms [18, 24, 36, 13] assume an extra condition not accounted for by Definition 3: that the number of concurrent operations has a fixed upper bound. An example of such implementation is shown in Figure 7a. This implementation of a concurrent wait-free queue has two uncommon features. First, besides adding and removing elements to the queue, it allows reading the head element without removing it. Second, it optimizes the memory usage by deallocating the head node when it is removed. Supporting both of these features, and simultaneously safety and linearizability is highly non-trivial. Therefore, the implementation in Figure 7a limits the concurrency to just two threads: an enqueuer thread that can either enqueue elements or read the queue head, and a dequeuer thread that can either dequeue elements or read the queue head.

Verifying such algorithms, which we call *restricted wait-free*, requires multiple changes to our approach as we have described so far. First of all, we define two wrapper functions `enqueuer` and `dequeuer` (see Figure 7b) which wrap the possible operations of the queue for the enqueuer and dequeuer threads explained above into a single operation respectively – each function checks its argument and dispatches the appropriate queue operation accordingly. Then, for technical reasons that we explain in Section 7, we restrict the queue to only store integers. To that end, the `enqueuer` wrapper includes a call to the **`to_int`** primitive (which we have added to $\lambda_{\text{ref}}^{\text{conc}}$ for this example), the semantics of which are as follows:

$$\texttt{to\_int}(n \in \mathbb{Z}) \overset{\text{pure}}{\rightsquigarrow} \texttt{Some}\, n \qquad \texttt{to\_int}(v \notin \mathbb{Z}) \overset{\text{pure}}{\rightsquigarrow} \texttt{None}$$

Restricted wait-freedom is stated as a variant of Definition 3, with changes highlighted:[6]

▶ **Definition 7** (Restricted wait-freedom). *Given a set of initialized configurations $C \subseteq Config$, we define restricted wait-freedom of list of operations* $\texttt{ops} : List(Val)$ *as*

$$\begin{array}{l} waitFreeRestr_C(\texttt{ops}) \triangleq \\ \quad \forall etr.\ length(etr[0].1) = length(\texttt{ops}) \wedge \\ \quad \left( \bigwedge_{\tau \mapsto e \,\in\, etr[0].1} \exists e'.\ etr[0].1[\tau] = e'[\texttt{ops}[\tau]/x] \wedge noLocs(e') \wedge noForks(e') \right) \wedge \\ \quad etr[0] \in C \Rightarrow \forall \tau.\ alwaysReturnsInTrace(\texttt{ops}[\tau], etr, \tau) \end{array}$$

This definition establishes a property of a *list* of methods – one operation may appear multiple times in this list. Initially (at the start of *etr*), we have *length*(`ops`) many threads, and the $i^{\text{th}}$ thread is a client of the $i^{\text{th}}$ method in `ops`. Importantly, none of the threads should fork any other threads – otherwise the restriction on the number of concurrent operations might be violated. The *alwaysReturnsInTrace* conclusion states that every thread's calls to its operation must always return.

For example, restricted wait-freedom of the algorithm in Figure 7b is stated using Definition 7 by taking $\texttt{ops} \triangleq [\texttt{enqueuer}, \texttt{dequeuer}]$. A more complicated $n$-enqueuer, single-dequeuer algorithm (also presented in [18]) requires to use a list consisting of $n$ `enqueuer` methods and a single `dequeuer` method, *i.e.*, $\texttt{ops} \triangleq \texttt{dequeuer} :: repeat(n, \texttt{enqueuer})$.

Verifying restricted wait-freedom requires changes to the specification we had given in Definition 4. We will first give the formal definition and explain the details afterwards.

▶ **Definition 8** (Lawyer specification of restricted wait-freedom). *For a set of initialized configurations $C \subseteq Config$, we define the Lawyer specification of* $\texttt{ops} : List(Val)$ *restricted wait-freedom as follows, for some* $\mathbf{F} : \mathbb{N}$*:*

$$\begin{array}{l} \mathsf{WaitFreeRestrSpec}_C(\texttt{ops}) \triangleq \forall c \in C.\ \mathsf{heap\_repr}(c) \Rrightarrow \mathsf{mtoks}(\texttt{ops}) * \\ \quad \ast_{\texttt{op} \in \texttt{ops}} \left( \forall \tau, \pi, a.\ \{\mathbf{F} \cdot \mathsf{bar}_\circ \pi * \mathsf{ph}_\circ \tau\, \pi * \mathsf{mtok}(\texttt{op})\}\ \texttt{op}\ a\ \{\mathsf{ph}_\circ \tau\, \pi * \mathsf{mtok}(\texttt{op})\}^{\tau} \right) * \\ \quad \ast_{\texttt{op} \in \texttt{ops}} \left( \forall \tau, a.\ \{\mathsf{mtok}(\texttt{op})\}^{\natural}\ \texttt{op}\ a\ \{v.\, isGroundVal(v) * \mathsf{mtok}(\texttt{op})\}^{\tau} \right) \\ \textit{where}\ \mathsf{mtoks}(l : List(Val)) \triangleq \mathsf{toks}_{\texttt{ops}}(l) \quad \textit{and} \quad \mathsf{mtok}(m : Val) \triangleq \mathsf{mtoks}([m]) \end{array}$$

[6] Note that below we assume that $\mathsf{Thread} \triangleq \mathbb{N}$ and that a thread pool can be treated as a list of expressions.

In the definition above, $\mathit{isGroundVal}(v : \mathit{Val})$ states that the value $v$ does not contain any locations or lambda functions. This is related to the restriction where we require the queue to only store integers which we will discuss in Section 7. The proposition $\mathsf{mtoks}(l)$ (an abbreviation for $\mathsf{toks}_{\mathsf{ops}}(l)$[7]) asserts ownership of the list $l$ of *method tokens*. Similarly to the tokens used in Section 3, these are created upon the module initialization and have to be provided upon every call to the corresponding operation; the difference is that a method token is given back after the return to allow subsequent calls. Method tokens satisfy the following laws similar to (tokens-sep) and (tokens-le); here, $\mathsf{count}(m, l)$ counts occurrences of $m$ in $l$:

$$\mathsf{mtoks}(l_1) * \mathsf{mtoks}(l_2) \dashv\vdash \mathsf{mtoks}(l_1 +\!+ l_2) \qquad \mathsf{mtoks}(l) \vdash \forall m \in l.\, \mathsf{count}(m, l) \leq \mathsf{count}(m, \mathsf{ops})$$

Finally, the adequacy theorem for restricted wait-freedom is similar to Theorem 5:

▶ **Theorem 9** (Adequacy for restricted wait-freedom). *If* $\mathsf{WaitFreeRestrSpec}_C(\mathtt{ops})$ *is provable in Lawyer, then* $\mathit{waitFreeRestr}_C(\mathtt{ops})$ *holds in the meta-logic.*

We use Theorem 9 to establish $\mathit{waitFreeRestr}_{C_Q}([\mathtt{enqueuer}, \mathtt{dequeuer}])$ (where $C_Q$ is some set of allowed initial states for the queue in Figure 7a) by verifying the wrappers in Figure 7b. We note that despite the fact that termination of queue methods follows almost immediately – they all terminate in constant time – proving their safety is highly non-trivial: reading of the head node might execute concurrently with dequeueing that involves freeing the head. We refer the reader to our Rocq development for the details of the proof.

## 6 Proving the Wait-Freedom Adequacy Theorem

As Section 4 demonstrates, proving the specification in Definition 4 for a given operation is relatively straightforward. Indeed, a wait-free operation cannot block, or be delayed by other operations running concurrently, thus it must terminate in finite number of steps. To account for this, the rules of Lawyer logic simply consume one fuel resource upon every step; essentially, the user is mainly responsible for providing enough fuel initially. This applies both to the original specification in Definition 4, and its variants described in Section 5. Thus, the non-trivial parts of the proof might only concern safety of the operation.

However, showing that the specification in Definition 4 implies wait-freedom, *i.e.* proving Theorem 5, is quite challenging. Ideally, we would just reuse the existing work on termination verification in Lawyer for proving wait-freedom. That is, we would ideally reuse Lawyer's proof that verifying a program in Lawyer establishes a refinement between the program and an always terminating transition system, which then guarantees that the program terminates. We might hope that this would help us derive termination of all calls to our wait-free operation. However, we face the following challenges:

**(C1)** The property established by Lawyer's adequacy theorem is termination of closed programs, *i.e.*, in our case the client program together with the wait-free data structure. In our case, we aim to prove termination of each individual call to the wait-free operation. The latter property would follow from the former, if the client program is terminating, but the client program does not have to be terminating in general.

[7] In fact, the tokens resource $\mathsf{toks}_N(k)$ from Figure 4 can be defined for any type of $k$ and $N$ that defines (disjoint) composition and the corresponding inclusion order.

**(C2)** Lawyer (and Trillium in general) requires the entire closed program to be verified for the adequacy theorem to apply, and to allow us to derive a property of its execution traces. For wait-freedom, we are given an arbitrary client program (that uses the wait-free operation) and cannot verify it directly.

**(C3)** The Lawyer logic does not contain rules that support verifying non-terminating programs. In particular, verifying such programs requires infinite amount of fuel, but there are no rules that allow generating fuel indefinitely long.

We address these challenges as follows. First, in Section 6.1 we reduce the problem of proving wait-freedom to showing termination of the closed program. With that, we are able to reuse the Lawyer's adequacy theorem, thus addressing the Challenge 1. Then, in Section 6.2 we employ the well-known method of logical relations to establish "robust safety" of the wait-free operation, *i.e.*, we prove that an arbitrary client does not violate operation's internal invariants. This solves the Challenge 2. Finally, in Section 6.3 we show that the aforementioned reduction in fact allows to apply the Lawyer logic to generate fuel consumed by an infinitely running program. Together with the robust safety result, it allows to verify a non-terminating program, which solves Challenge 3.

Finally, we note that Theorem 5 and its variants for possibly-stuck wait-freedom from Section 5.1 are proven in the same way with minimal variations. However, proving Theorem 9 requires non-trivial changes to the logical relation and the way it is used. For space reasons, we refer the reader to technical Rocq development for that. Therefore, the rest of the section only explains how the original Theorem 5 is proven.

## 6.1 Reducing Wait-Freedom to Termination

The first step in proving the wait-freedom adequacy theorem is to reduce proving wait-freedom to showing termination of the closed program. For that, the proof of Theorem 5 proceeds via a proof by contradiction. Namely, for the operation $\mathtt{op}$ and some set of initialized configurations $C \subseteq \mathit{Config}$ we assume that an execution trace $\mathit{etr}$, evaluation context $E$, thread $\tau$ and trace index $i$ such that:

- $\mathsf{WaitFreeSpec}_C(\mathtt{op})$ is provable in Lawyer.
- The premises of both $\mathit{waitFree}_C(\mathtt{op})$ and $\mathit{alwaysReturnsInTrace}(\mathtt{op}, \mathit{etr}, \tau)$ hold. In particular, a call to $\mathtt{op}$ occurs in $\mathit{etr}$ at index $i$, thread $\tau$, under evaluation context $E$.
- Yet, the conclusion of $\mathit{alwaysReturnsInTrace}(\mathtt{op}, \mathit{etr}, \tau)$ does not hold, *i.e.* $\neg(\exists j \geq i.\, \mathit{Ret}(\mathit{etr}[j].1, \tau, E))$.

Note that the only way this is possible is in the case where $\mathit{etr}$ is infinite. Indeed, if the call to $\mathtt{op}$ never returns, then by the *schedUntilRet* assumption of *alwaysReturnsInTrace* the calling thread $\tau$ is always eventually scheduled. We will prove the contradiction by showing that the same set of assumptions simultaneously implies that $\mathit{etr}$ must terminate.

To prove that $\mathit{etr}$ terminates, we refine it to a trace of the Obligations Model of Lawyer; the adequacy theorem of Lawyer implies the finiteness of both traces. For establishing the refinement, we need to verify the program (*i.e.*, each element of the expressions in the starting thread pool, $\mathit{etr}[0].1$). As mentioned in Challenge 2 and Challenge 3, the Lawyer logic can only support terminating programs, whereas we already concluded above that $\mathit{etr}$ is infinite, and moreover we cannot apply the rules of Lawyer directly, as the program to verify (a client of the wait-free operation) is universally quantified over.

The key to establishing the refinement is exploiting the assumptions above. Indeed, in Section 6.3 we show how to reuse Lawyer logic and construct a refinement *assuming* the premises above and given the specification for the wait-free operation. In other words, we

establish the refinement *only* for execution traces that satisfy the premises above. This is in contrast with the adequacy theorem of Lawyer (and Trillium) which establish the refinement for *all* execution traces. Therefore, we need to generalize the adequacy theorem of Trillium to establish a refinement only when some condition on execution holds. In return, the fact that this condition holds is propagated to the refinement construction via user-provided "progress resource" described in more detail in Section 6.3. Below, we provide the generalized adequacy theorem of Trillium and highlight the main changes compared to the original theorem.

▶ **Theorem 10** (Conditional adequacy theorem of Trillium). *Consider a transition system $\mathcal{M}$ and a relative image-finite relation $\xi$ on finite traces of $\lambda_{\mathsf{ref}}^{\mathsf{conc}}$ and the model. Then, consider* $\mathsf{TI} : FinTrace(\lambda_{\mathsf{ref}}^{\mathsf{conc}}) \to FinTrace(\mathcal{M}) \to iProp$ *– a representation of the aforementioned finite traces in the Iris logic. Then, consider a predicate $F$ on finite execution traces. Finally, consider a "progress resource"* $\mathsf{PR}_F : FinTrace(\lambda_{\mathsf{ref}}^{\mathsf{conc}}) \to iProp$ *that satisfies a number of laws described in appendix [30].*

*Let etr be an execution trace and* $\mathsf{m}$ *a state of $\mathcal{M}$. If $\xi(etr[0], \mathsf{m})$ holds (treating its arguments as singleton traces), $F(etr')$ holds for any finite prefix $etr'$ of etr, and furthermore we have (again, treating $etr[0]$ and* $\mathsf{m}$ *as singleton traces)*

$$\vdash \Rrightarrow \mathsf{TI}(etr[0], \mathsf{m}) * \mathsf{AlwaysHolds}(\xi, etr[0], \mathsf{m}) * \mathsf{PR}_F(etr[0])$$

*then there exists such model trace mtr that $mtr[0] = \mathsf{m}$ and $\hat{\xi}(etr, mtr)$ holds.*

We refer the reader to Trillium [35] for definitions of $\mathsf{AlwaysHolds}$ and $\hat{\xi}$; the latter is essentially the refinement we are looking for. Intuitively, the three resources in the Iris premise of Theorem 10 correspondingly describe *what* is the current state of the execution, *why* this state representation implies refinement at the meta-level, and *how* this representation is preserved on every step of the execution.

We prove the finiteness of our *etr* by applying Theorem 10 and establishing the refinement to the Obligations Model. For that, we choose the trace interpretation, the starting model state $\mathsf{m}$, and the refinement relation that the Lawyer's adequacy theorem does when it uses Trillium's adequacy theorem (the latter being slightly adjusted – see our Rocq implementation for details). We defer the choice of the predicate $F$ and the progress resource to Section 6.3; we only note that the chosen $F$ indeed holds for all prefixes of *etr*. Thus, it remains to prove the Iris premise of Theorem 10. There, $\mathsf{TI}(etr[0], \mathsf{m})$ and $\mathsf{AlwaysHolds}(\xi, etr[0], \mathsf{m})$ are proved similarly to how it is done in Lawyer. We will explain how to establish the progress resource in Section 6.3. Thus, using the Trillium's adequacy theorem we prove that *etr* is finite, and thus derive a contradiction.

## 6.2 Logical Relation for Ensuring "Robust Safety" of a Wait-Free Operation

As the Challenge 2 states, using a Lawyer-based approach requires us to verify the entire program under consideration, including the arbitrary and thus unknown client of the operation. Moreover, as the Challenge 3 states, the Lawyer logic in general does not support verification of non-terminating programs which we would like to consider as clients of wait-free operations.

However, it turns out that verifying the operation's client can be split into two steps. The first step is proving the *robust safety* of the operation, *i.e.* the fact that any operation's client preserves operation's internal invariants. The next step, described further in Section 6.3, is showing that any execution of a client refines the Obligations Model of Lawyer.

To prove robust safety, we employ the well-known idea of logical relations [34, 12]. This approach is based on proving, for a given expression $e$, an Iris proposition $\mathsf{RS}_{\mathbf{E}}(e)$; the predicate $\mathsf{RS}_{\mathbf{E}}(\cdot)$ is called the *expression relation*. The expression relation intuitively captures

that any execution of $e$ preserves all invariants in the system (including those of the wait-free operation), but it may get stuck. Crucially, the expression relation is proven to be closed under program compositions, *e.g.*, if both $e_1$ and $e_2$ are in the expression relation, so is the expression $e_1\ e_2$, *i.e.*, calling the function $e_1$ with argument $e_2$ – intuitively, the worst that can happen here is $e_1$ not a function in which case $e_1\ e_2$ gets stuck, which is allowed. This compositionality of the expression relation is the reason why to establish robust safety of the operation it is sufficient to prove $\mathsf{RS}_{\mathbf{E}}(e)$ for all clients $e$ of the operation. In Section 4 we described a PresInv Hoare triple that states a property similar to $\mathsf{RS}_{\mathbf{E}}(\cdot)$. Indeed, expression relation is defined similarly to PresInv (we explain $\mathsf{RS}_{\mathbf{V}}(v)$ below):

$$\mathsf{RS}_{\mathbf{E}}(e) \triangleq \forall \tau.\ \{\mathsf{True}\}^{\natural}\ e\ \{v.\mathsf{RS}_{\mathbf{V}}(v)\}^{\tau}$$

To understand how $\mathsf{RS}_{\mathbf{E}}(e)$ is proven, first consider some $e$ that does not actually call the wait-free operation. The potentially unsafe steps of $e$ execution are writing to memory location (or freeing it) or evaluating an arbitrary, potentially unsafe lambda expression. For these steps to be safe, we need to ensure that the corresponding location or lambda expression are also "safe to use". More generally, we introduce the following *value relation* $\mathsf{RS}_{\mathbf{V}}(v)$:[8]

▶ **Definition 11** (Value relation)**.**

$$\begin{aligned}
\mathsf{RS}_{\mathbf{V}}(()) \triangleq \mathsf{RS}_{\mathbf{V}}(b:\mathbb{B}) &\triangleq \mathsf{RS}_{\mathbf{V}}(n:\mathbb{Z}) \triangleq \mathsf{True}\\
\mathsf{RS}_{\mathbf{V}}(\mathbf{inl}\ v) &\triangleq \mathsf{RS}_{\mathbf{V}}(\mathbf{inr}\ v) \triangleq \mathsf{RS}_{\mathbf{V}}(v)\\
\mathsf{RS}_{\mathbf{V}}((v_1,v_2)) &\triangleq \mathsf{RS}_{\mathbf{V}}(v_1) * \mathsf{RS}_{\mathbf{V}}(v_2)\\
\mathsf{RS}_{\mathbf{V}}(\ell : \mathit{Loc}) &\triangleq \boxed{(\exists v : \mathit{Val}.\ \ell \mapsto v * \mathsf{RS}_{\mathbf{V}}(v)) \vee \mathsf{freed}(\ell)}\\
\mathsf{RS}_{\mathbf{V}}(\mathbf{rec}\ f\ x := e) &\triangleq \forall \tau, v.\ \{\mathsf{RS}_{\mathbf{V}}(v)\}^{\natural}\ (\mathbf{rec}\ f\ x := e)v\ \{r.\,\mathsf{RS}_{\mathbf{V}}(r)\}^{\tau}
\end{aligned}$$

Note that the last two cases of this definition reference $\mathsf{RS}_{\mathbf{V}}(v')$ for some value $v'$ which might be structurally larger than one being considered. The soundness of such definitions is guaranteed by the guarded recursion of Iris; we refer the reader to [20] for more details.

Thus, proving $\mathsf{RS}_{\mathbf{E}}(e)$ for an expression $e$ that does not call the wait-free operation involves keeping track of the values produced by executing $e$ so far and making sure they all satisfy the value relation. With that, if $e$ does not contain hard-coded locations initially and does not use pointer arithmetic, we can indeed prove $\mathsf{RS}_{\mathbf{E}}(e)$ by structural induction[9], showing that evaluating its sub-expressions produce values that can be safely used afterwards.

However, an $e$ that actually calls a wait-free operation `op` requires some extra work. Indeed, values produced during `op` execution might not satisfy the value relation (*e.g.* locations used by `op` internally are covered by `op`'s own invariants), and also `op` might use pointer arithmetic (*e.g.* queue in Figure 7a uses it for accessing `Node` structure fields). Thus, we cannot automatically derive expression relation for calls to `op`– instead, the user must show it. And indeed, the PresInv(`op`) part of `op` wait-free specification is exactly the lambda-expression case of Definition 11, stating that it is always safe to call `op` with any argument. Another way to think about this is that showing PresInv(`op`) is also required because it ensures that the operation does not expose its own internal state, which would of course not be safe.

A more general fact is stated by the "fundamental theorem" of logical relations. Namely, that the expression relation holds for any expressions whose free variables are substituted with values that are themselves in the value relation:

---

[8] The resource $\mathsf{freed}(\ell)$ is obtained after executing $\mathbf{free}\,\ell$ and is irrelevant for the rest of the paper.

[9] For lambda expressions, it also involves so-called Löb induction available in Iris-like step-indexed logics.

▶ **Theorem 12** (Fundamental theorem). *Let $e$ be an expression that contains neither any Loc literals nor any pointer arithmetic, and whose free variables are $\overrightarrow{xs}$. Given any list of values $\overrightarrow{vs}$ such that $|\overrightarrow{vs}| = |\overrightarrow{xs}|$. Then the following holds:*

$$\left(\ast_{v \in \overrightarrow{vs}} \mathsf{RS}_{\mathbf{V}}(v)\right) \vdash \mathsf{RS}_{\mathbf{E}}(e[\overrightarrow{vs}/\overrightarrow{xs}])$$

An op's client can be represented as a thread pool made of expressions into which op is substituted (see the first premise of Definition 3). To prove robust safety of op, we apply Theorem 12 and provide $\mathsf{PresInv}(\mathsf{op})$ to satisfy its premise. Thus, we establish the following:

▶ **Theorem 13** (Robust safety of the wait-free operation). *Let $\mathcal{E}$ be a thread pool such that its every element $e$ has a single free variable $x$, and contains neither Loc literals, nor any pointer arithmetic. Assume that $\mathsf{WaitFreeSpec}_C(\mathsf{op})$ holds for some $C$. Then the following holds:*

$$\vdash \ast_{e \in \mathcal{E}} \mathsf{RS}_{\mathbf{E}}(e[\mathsf{op}/x])$$

## 6.3 Refining an Infinite Execution to a Trace of Obligations Model

In Section 6.1, we have reduced proving wait-freedom to that of termination, which we establish by applying Theorem 10 and refining a given execution to a trace of Obligations Model of Lawyer. However, that reduction, by contradiction, considers an *infinite* execution, whereas the main property of Obligations Model is the finiteness of its traces. As the Challenge 3 states, the Lawyer logic that establishes refinement to the Obligations Model was designed to only verify terminating programs. Here, we briefly describe the solution that allows to repurpose the Lawyer logic for verifying non-terminating programs. Due to lack of space we refer the reader to our Rocq formalization [29] for the details.

The central part in establishing refinement with Theorem 10 is the progress resource. Intuitively, the resource $\mathsf{PR}_F(etr)$ means that if the finite execution trace $etr$ is currently refined by some trace of the Obligations Model, and if it takes a step into a new trace $etr'$ such that $F(etr')$ holds, then $etr'$ will also refine some Obligations Model trace, all the invariants are preserved, and $\mathsf{PR}_F(etr')$ is reestablished. Thus, the progress resource for a singleton trace $\mathsf{PR}_F(c)$ essentially means that any execution starting from the configuration $c$ is refined by a trace of the Obligations Model, as long as it does not violate the predicate $F$.

Crucially, the user is free to choose an arbitrary progress resource, as long as it satisfies a number of laws listed in the appendix [30]. The original adequacy theorem of Lawyer can be stated as a corollary of Theorem 10 where the progress resource ignores the predicate $F$, and for the starting configuration it consists of Hoare triples for every thread present in that configuration. In our case, we cannot prove such triples, as we consider non-terminating client programs, and the Lawyer logic is designed to establish termination.

Thus, the progress resource for wait-freedom consists of two parts: one that guarantees preservation of invariants and another that ensures the refinement of the Obligations Model. The former is given by the robust safety result of Theorem 13; the latter is more elaborate.

The main part for establishing a refinement of the Obligations Model is providing a barrel of fuel on every execution step. For that, we exploit the assumptions on the execution that were obtained during the reduction in Section 6.1. Namely, that our progress resource is defined as $F(etr) \triangleq \mathit{infCallPrefix}(E, \tau, i, etr)$, where $E, \tau$ and $i$ are fixed during the reduction. We refer the reader to Rocq development [29] for its formal definition. Intuitively, this predicate means that $etr$ is a possible finite prefix of some infinite trace in which thread

$\tau$ has an infinite call to `op` under evaluation context $E$ at index $i$. Now, knowing $i$, we can pre-allocate $i$ barrels of fuel in the initial progress resource, so that the refinement can be preserved for the first $i$ steps before the infinite call.

To provide fuel for the rest of execution, we first note that the thread $\tau$ that runs that call does not need infinite fuel: the specification NoInfExec(`op`) states that any call to `op` refines a trace of the Obligations Model, as long as a fixed amount of fuel is provided to satisfy the precondition of that specification. Therefore, we add this fuel to the initial progress resource.

To provide fuel to all other threads, we employ a trick that allows to *generate fuel indefinitely*. That is, we use the Lawyer logic's obligations mechanism originally designed for verifying blocking concurrency. We refer the reader to Lawyer [31] for more details; in short, threads can wait for other threads' actions by generating fuel, as long as this waiting is not circular. In our case, we can assign the calling thread $\tau$ an obligation that is never fulfilled which does not correspond to any action, but merely enables the fuel generation mechanism. This lets all other threads to wait for this obligation to be fulfilled. By assumption, $\tau$ never terminates, and thus the other threads can always receive fuel. Of course, this trick is only possible because we started the proof in Section 6.1 by contradiction – under normal circumstances, a thread with an obligation eventually fulfills it, which disables fuel generation.

# 7 Discussion and Limitations

## 7.1 Restricting the content of wait-free data structures

When verifying the queue implementation in Section 5.2, we introduced a seemingly unnecessary constraint. Namely, the wrapper `enqueuer` in Figure 7b only allows the integer values to be enqueued. Note that the original implementation in Figure 7a never inspects the enqueued values and thus does not rely on this constraint. Yet, in short, lifting this constraint would imply adding an extra premise to the precondition of NoInfExec which must hold for the argument – but since we show termination of calls to an operation with *any* argument, we would not be able to prove it when using NoInfExec in the adequacy proof.

Before proceeding to details of this constraint, we explain its scope. Despite arising when verifying restricted wait-freedom in Section 5.2, the issue is not related to tokens, or the restricted wait-freedom in general. However, it arises with all data structures that contain methods that return values previously stored by other methods (queues, stacks, linked lists *etc.*). In our context, such structures happen to be restricted wait-free.

Recall that establishing robust safety of `dequeuer` in Figure 7a using Theorem 13 requires proving the PresInv(`dequeuer`) specification, which is the lambda-expression case of Definition 11. (For technical reasons, Definition 8 requires a stronger specification from which PresInv(`dequeuer`) is derivable, but the issue would still hold without this stronger specification.) Note that when `dequeuer` returns a dequeued value, it cannot, by itself, prove that it is safe-to-use. For this to hold, the queue invariant must enforce that all values stored are safe-to-use by storing the $\mathsf{RS}_{\mathbf{V}}(v)$ resources for every stored value $v$. To maintain this invariant, `enqueuer` must only call `enqueue` with safe-to-use values – and it indeed does, as only integers are being enqueued.

If we remove the only-integers constraint, the specifications for `enqueuer` would have to include the $\mathsf{RS}_{\mathbf{V}}(v)$ premise in the precondition. In particular, it would appear in the precondition of NoInfExec(`enqueuer`). When proving wait-freedom, we consider a potentially non-terminating call to `enqueuer` with some argument $a$. Internally, the adequacy theorem uses (the token-based version of) NoInfExec(`enqueuer`) to verify the call `enqueuer` $a$. Since $\mathsf{RS}_{\mathbf{V}}(a)$ is not provable in general for arbitrary $a$, we would not be able to use this specification.

To lift this limitation, one has to exploit the fact that a client program satisfying the conditions of Theorem 13 can only produce safe-to-use values as it executes (except when the wait-free operation is in progress). We leave establishing this formally to future work.

## 7.2 Proving linearizability of wait-free data structures

An important criteria for the correctness of a concurrent data structure is *linearizability* [16]. A data structure is linearizable if whenever operations on that structure are run concurrently, each of them appear to take effect atomically, even if they actually take multiple steps to finish. Linearizability makes it easier to reason about the data structure as one does not have to consider all possible interleavings of the operations – one can simply reason as though these operations happen one after another. It is thus not surprising that many wait-free data structures are designed to be linearizable [18, 24, 36, 13]. Therefore, it is desirable to verify both wait-freedom and linearizability for wait-free data structure.

In fact, proving linearizability with program logics is well-studied [17, 7, 22, 3]. Birkedal *et al.* [3] have formally shown that linearizability is internalized by specifications given in terms of the so-called *logically atomic Hoare triples* [22]. Moreover, the Lawyer approach [31] that we build upon provides a liveness-aware version of the logically atomic specification. However, even simple wait-free implementations contain what is known as *future-dependent linearization points*[37] where the point at which linearization takes place within an operation can depend on how the other operations started afterwards execute. Moreover, the linearization point of an operation might happen inside another operation that runs concurrently with it in a different thread. It is the case for the queue in Figure 7a. Consider the scenario where the dequeuer thread runs a number of `dequeue` operations one by one, while the enqueuer thread is running the single `read_head_e` operation. In that case, the linearization point of `read_head_e`, if the comparison after the second read of `Head` fails, will occur *when the last* `dequeue` *operation writes to* `OldHeadVal`.

Iris supports verifying future-dependent linearizability with using so-called prophecy variables [21]. However, as is, Lawyer does not support prophecy variables because Trillium [35] does not. Thus, verifying both wait-freedom and linearizability in our framework would require extending the Lawyer logic and the Trillium logic first.

## 7.3 Wait-free operations of a lock-free data structure

Some data structures provide both lock-free and wait-free methods. For example, an implementation of counter can include a lock-free increment from Figure 1b and a wait-free `read` method that simply reads the current counter value in a single step. Thus, in general, wait-freedom is a property of a single method, not an entire data structure. Our approach could also be used for verifying wait-freedom in such a setting as well. First, the meta-level definition of wait-freedom should mention both a wait-free operation, and a number of lock-free operations that may run concurrently with it (similarly to how Definition 7 applies to the list of wait-free operations). As for the specifications, the approach suggested by Total TaDA [8] applies: the operations running concurrently with the wait-free one only need to preserve invariants. With that, we can prove a generalization of Theorem 5 that establishes wait-freedom of an operation `op` given $\mathsf{WaitFreeSpec}(\texttt{op})$ and proofs of $\mathsf{PresInv}(\texttt{op}')$ for every lock-free operation $\texttt{op}'$ running concurrently with `op`.

# 8 Related Work

All prior solutions verify wait-freedom for first-order languages. These solutions assume, in one way or another, that the client of a wait-free operation is safe, *i.e.* that it does not violate the data structure's internal invariants. Instead, we prove once and for all that all possible clients in our higher-order programming language satisfy this property via a logical relations argument as explained in Section 6.2. Moreover, most of the prior solutions only support unrestricted wait-freedom, lack case studies, and are not mechanized.

## 8.1 Program Logics for Wait-Freedom

**Total TaDA.** The program logic of [8] supports verification of non-blocking program properties, including unrestricted wait-freedom. Proving a program specification in that logic establishes a meta-level judgement about program execution. This judgement is defined as a least fixed point and thus guarantees the program's termination. The non-trivial (from liveness perspective) rules of the Total TaDA logic allow verification of while loops and recursive functions by choosing a decreasing measure of termination. Moreover, their specifications are capable of expressing linearizability, which we do not consider. This approach is demonstrated by verifying wait-freedom of `read` operation of a counter.

Total TaDA assumes that the operation's client is safe by employing a rely relation [19] into the aforementioned meta-level judgement established for the operation. The only way to ensure that the client respects the rely relation is to verify it; thus their definition of wait-freedom does not quantify over arbitrary client as ours does. However, they support clients verified with the non-total TaDA [7] logic. Contrary to that, our approach supports arbitrary client programs (up to assumptions of Theorem 12). Total Tada is not mechanized.

**Lili.** The approach of [26] verifies liveness of both blocking and non-blocking operations of concurrent objects, as well as their linearizability. In that work, (unrestricted) wait-freedom is treated as starvation-freedom (a property usually applied to blocking algorithms) without the fair scheduling assumption. The Lili logic is parametric, and by instantiating it in a specific way one can exclude rules that justify threads being delayed and/or blocked by each other. This is similar to how Lawyer proves *e.g.* termination under unfair scheduler with a more restrictive instantiation of Obligations Model.

The adequacy statement of Lili is similar to ours. It states that for an arbitrary execution trace starting from an arbitrary client program configuration, and an "appropriate" state, all calls to the object's methods terminate. However, the major difference compared to our approach is that the physical state of the concurrent object and its client are explicitly separated. That way, in Lili the invariants of the concurrent object are trivially preserved by its clients who do not have access to them at all. In our work, our higher-order language does not allow us easily state and use such a separation, and thus we use our logical relations model to prove safety of arbitrary client.

The Lili logic is neither higher-order nor mechanized. Moreover, neither Lili, not the lock-free fragment of its predecessor [27] have any case studies on wait-freedom verification.

**Automatic tool for verifying non-blocking algorithms.** The work of [14] verifies progress properties of non-blocking algorithms. They notice that progress of these algorithms can be established by identifying a set of ordered abstract actions. An action at a higher level cannot be executed infinitely often unless there is an action of lower level which is also executed infinitely often. In this setting, (unrestricted) wait-freedom corresponds to discrete

orders, *i.e.* no action can ever be executed infinitely often. In the absence of non-trivial orders between actions and unbounded `while` loops, Gotsman *et al.* [14] reduce verifying wait-freedom to verifying safety.

This solution relies on an automatic tool that establishes safety of a given operation. Moreover, it establishes the rely relation that the operation's client is expected to preserve. Similarly to Total TaDA, this means that wait-freedom is only established for verified clients.

Gotsman *et al.* [14] present no wait-freedom case studies. They explain that their approach might not apply to non-trivial wait-free operations due to the limitations of their tool. As an example, they mention the wait-free `contains` operation of the concurrent set algorithm by [38]. The `while` loop in that operation traverses a sorted list of integers up to a given number. As the list can be concurrently modified during the traversal, termination of such loop relies on the list being sorted. The tool by [14] cannot exploit this fact, whereas in our approach the termination of such loops can be shown by providing the amount of fuel proportional to the given number.

Moreover, their tool has limited safety reasoning capabilities and thus cannot verify examples like Figure 7a where the safety argument is highly non-trivial. The tool is automated, but the presented theory is not mechanized in a proof assistant.

**Wait-freedom of "fast-path-slow-path" in VST.** The work of [32] considers non-blocking algorithms that are instances of a specific variation of the "fast-path-slow-path" approach [24]. This approach aims to design efficient wait-free data structures by combining a lock-free implementation (which is often faster, but does not have wait-free progress guarantee) and a wait-free implementation (slower, but has guaranteed progress).

The authors of [32] establish a pen-and-paper proof that such algorithms, if safe, are indeed wait-free, and provide an extension of the VST logic [1] to verify their safety, and thus also wait-freedom. This approach limits the number of concurrent threads, and therefore it can only be used to establish restricted wait-freedom. Similar to Total TaDA and Gotsman *et al.* [14], such approach requires the user to verify the entire program.

As a case study, Peterson *et al.* [32] verify wait-freedom of the queue by Kogan and Petrank [24], which is in turn an optimization of the queue by Kogan and Petrank [23]. In the implementation by [24], a thread helps up to $N$ other concurrent operations before performing its own operation (where $N$ is the maximal number of threads). We believe that our approach can also support verifying this case study by requiring an amount of fuel proportional to $N$.

## 8.2 Universal Constructions

Designing a wait-free implementation for every possible data structure is tedious, and therefore a question arises as to whether it is possible to design a universal wait-free construction. And indeed, that is possible. These are algorithms that turn any *sequential* data structure implementation into a (restricted) wait-free one. The early universal constructions such as [15] are considered impractical due to their high time and space complexity of the resulting wait-free implementations. Luckily, this is less of an issue for more recent constructions mentioned below. However, none of these constructions has a mechanized proof of wait-freedom.

Therefore, it is interesting whether our solution can be used to verify such constructions. To do so, we would need to prove a specification for the sequential implementation, and use it together with a general proof of the universal construction in question – a process similar to how we verified possibly-stuck wait-freedom of `list_map` (`incr` $l$) in Section 5.1. Therefore, we believe that our approach can be applied to universal constructions as well.

Examples of modern wait-free universal constructions include P-Sim [10] (along with its extensions such as [11]) and CX [6]. The time complexity of these approaches is proportional to the number of concurrent operations, and the execution time of the provided sequential implementation. Given that these parameters are known, we believe that our approach can verify restricted wait-freedom of the resulting implementations, as the required amount of fuel is known in advance.

## 9 Conclusion and Future Work

In this work, we have presented the first solution to verifying wait-freedom in a higher-order language. The key idea is internalizing the notion of wait-freedom in the Lawyer separation logic with a particular specification pattern. To establish this connection formally we presented a new adequacy theorem for Lawyer specifically for programs proven against a specification in the aforementioned pattern. We also used a logical relations model to prove that a wait-free operation can be safely used by an arbitrary, higher-order client. We demonstrated our approach by verifying a number of examples, with the more complicated of them requiring a slightly different notion of wait-freedom which we call restricted wait-freedom, and to accordingly adjust the wait-freedom specification pattern. Our solution and case studies are all mechanized in the Rocq Prover.

There are multiple notable directions of future work. First, we aim to extend our approach to verify related properties, such as linearizability and bounded wait-freedom [5]. It would also be interesting to verify one of the state-of-the-art universal constructions for wait-freedom. Finally, we hope to be able to improve the usability of our approach by deriving the two specifications needed by Definition 4 from a single, stronger specification that implies both.

## References


1 Andrew W. Appel. Verified software toolchain. In Gilles Barthe, editor, *Programming Languages and Systems*, pages 1–17, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

2 Lars Birkedal and Ales Bizjak. Lecture notes on Iris: Higher-order concurrent separation logic, 2017. URL: `http://iris-project.org/tutorial-pdfs/iris-lecture-notes.pdf`.

3 Lars Birkedal, Thomas Dinsdale-Young, Armaël Guéneau, Guilhem Jaber, Kasper Svendsen, and Nikos Tzevelekos. Theorems for free from separation logic specifications. *Proc. ACM Program. Lang.*, 5(ICFP), August 2021. `doi:10.1145/3473586`.

4 Fabian Bläse. Non-blocking synchronization in operating systems, 2021. URL: `https://www4.cs.fau.de/Lehre/WS20/MS_AKSS/arbeiten/paper_blaese_final.pdf`.

5 Hagit Brit and Shlomo Moran. Wait-freedom vs. bounded wait-freedom in public data structures (extended abstract). In James H. Anderson, David Peleg, and Elizabeth Borowsky, editors, *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing, Los Angeles, California, USA, August 14-17, 1994*, PODC '94, pages 52–60, New York, NY, USA, 1994. ACM. `doi:10.1145/197917.197950`.

6 Andreia Correia, Pedro Ramalhete, and Pascal Felber. A wait-free universal construction for large objects. In Rajiv Gupta and Xipeng Shen, editors, *PPoPP '20: 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California, USA, February 22-26, 2020*, PPoPP '20, pages 102–116, New York, NY, USA, 2020. ACM. `doi:10.1145/3332466.3374523`.

7 Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. Tada: A logic for time and data abstraction. In Richard E. Jones, editor, *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*, Lecture Notes in Computer Science, pages 207–231. Springer, 2014. `doi:10.1007/978-3-662-44202-9_9`.

8 Pedro da Rocha Pinto, Thomas Dinsdale-Young, Philippa Gardner, and Julian Sutherland. Modular termination verification for non-blocking concurrency. In Peter Thiemann, editor, *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9632 of *Lecture Notes in Computer Science*, pages 176–201. Springer, 2016. doi:10.1007/978-3-662-49498-1_8.

9 Derek Dreyer, Georg Neis, and Lars Birkedal. The impact of higher-order state and control effects on local relational reasoning. In Paul Hudak and Stephanie Weirich, editors, *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 143–156. ACM, 2010. doi:10.1145/1863543.1863566.

10 Panagiota Fatourou and Nikolaos D. Kallimanis. A highly-efficient wait-free universal construction. In Rajmohan Rajaraman and Friedhelm Meyer auf der Heide, editors, *SPAA 2011: Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Jose, CA, USA, June 4-6, 2011 (Co-located with FCRC 2011)*, SPAA '11, pages 325–334, New York, NY, USA, 2011. ACM. doi:10.1145/1989493.1989549.

11 Panagiota Fatourou, Nikolaos D. Kallimanis, and Eleni Kanellou. An Efficient Universal Construction for Large Objects. In Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller, editors, *23rd International Conference on Principles of Distributed Systems (OPODIS 2019)*, volume 153 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:15, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.OPODIS.2019.18.

12 {Aïna Linn} Georges. Designing and proving robust safety of efficient capability machine programs, July 2023.

13 Seep Goel, Pooja Aggarwal, and Smruti R. Sarangi. A wait-free stack. In Nikolaj S. Bjørner, Sanjiva Prasad, and Laxmi Parida, editors, *Distributed Computing and Internet Technology - 12th International Conference, ICDCIT 2016, Bhubaneswar, India, January 15-18, 2016, Proceedings*, Lecture Notes in Computer Science, pages 43–55. Springer, 2016. doi:10.1007/978-3-319-28034-9_6.

14 Alexey Gotsman, Byron Cook, Matthew J. Parkinson, and Viktor Vafeiadis. Proving that non-blocking algorithms don't block. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 16–28. ACM, 2009. doi:10.1145/1480881.1480886.

15 Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991. doi:10.1145/114005.102808.

16 Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990. doi:10.1145/78969.78972.

17 Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, POPL '11, pages 271–282, New York, NY, USA, 2011. ACM. doi:10.1145/1926385.1926417.

18 Prasad Jayanti and Srdjan Petrovic. Logarithmic-time single deleter, multiple inserter wait-free queues and stacks. In Ramaswamy Ramanujam and Sandeep Sen, editors, *FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science, 25th International Conference, Hyderabad, India, December 15-18, 2005, Proceedings*, Lecture Notes in Computer Science, pages 408–419. Springer, 2005. doi:10.1007/11590156_33.

19 Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, October 1983. doi:10.1145/69575.69577.

20 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28:e20, 2018. doi:10.1017/S0956796818000151.

21 Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. The future is ours: prophecy variables in separation logic. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. doi:10.1145/3371113.

22 Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 637–650. ACM, 2015. doi:10.1145/2676726.2676980.

23 Alex Kogan and Erez Petrank. Wait-free queues with multiple enqueuers and dequeuers. *SIGPLAN Not.*, pages 223–234, 2011. doi:10.1145/1941553.1941585.

24 Alex Kogan and Erez Petrank. A methodology for creating fast wait-free data structures. *SIGPLAN Not.*, pages 141–150, 2012. doi:10.1145/2145816.2145835.

25 Pierre LaBorde, Steven D. Feldman, and Damian Dechev. A wait-free hash map. *Int. J. Parallel Program.*, 45(3):421–448, June 2017. doi:10.1007/s10766-015-0376-3.

26 Hongjin Liang and Xinyu Feng. A program logic for concurrent objects under fair scheduling. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, POPL '16, pages 385–399, New York, NY, USA, 2016. ACM. doi:10.1145/2837614.2837635.

27 Hongjin Liang, Xinyu Feng, and Zhong Shao. Compositional verification of termination-preserving refinement of concurrent programs. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2603088.2603123.

28 Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In James E. Burns and Yoram Moses, editors, *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, USA, May 23-26, 1996*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM. doi:10.1145/248052.248106.

29 Egor Namakonov, Lars Birkedal, and Amin Timany. Verifying wait-freedom for concurrent higher-order programs (artifact), April 2026. URL: https://zenodo.org/records/19607892/files/artifact.pdf.

30 Egor Namakonov, Lars Birkedal, and Amin Timany. Verifying wait-freedom for concurrent higher-order programs (technical appendix), April 2026. URL: https://zenodo.org/records/19607892/files/paper-appendix.pdf.

31 Egor Namakonov, Justus Fasse, Bart Jacobs, Lars Birkedal, and Amin Timany. Lawyer: Modular obligations-based liveness reasoning in higher-order impredicative concurrent separation logic. *Proc. ACM Program. Lang.*, 10(OOPSLA1), April 2026. doi:10.1145/3798240.

32 Christina Peterson, Victor Cook, and Damian Dechev. Practical progress verification of descriptor-based non-blocking data structures. In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 83–93, 2019. doi:10.1109/MASCOTS.2019.00019.

33 Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. Transfinite iris: resolving an existential dilemma of step-indexed separation logic. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 80–95. ACM, 2021. doi:10.1145/3453483.3454031.

**34** David Swasey, Deepak Garg, and Derek Dreyer. Robust and compositional verification of object capability patterns. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017. doi:10.1145/3133913.

**35** Amin Timany, Simon Oddershede Gregersen, Léo Stefanesco, Jonas Kastberg Hinrichsen, Léon Gondelman, Abel Nieto, and Lars Birkedal. Trillium: Higher-order concurrent and distributed separation logic for intensional refinement. *Proc. ACM Program. Lang.*, 8(POPL), January 2024. doi:10.1145/3632851.

**36** Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. Wait-free linked-lists. In J. Ramanujam and P. Sadayappan, editors, *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012*, pages 309–310. ACM, 2012. doi:10.1145/2145816.2145869.

**37** Viktor Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, UK, July 2008. doi:10.48456/tr-726.

**38** Martin T. Vechev and Eran Yahav. Deriving linearizable fine-grained concurrent objects. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 125–135. ACM, 2008. doi:10.1145/1375581.1375598.