Verifying wait-freedom for concurrent higher-order programs

Anonymous author
Anonymous affiliation

Anonymous author
Anonymous affiliation

Anonymous author
Anonymous affiliation

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 extend Lawyer with a novel adequacy theorem that proves that programs which are proven correct in the Lawyer logic gainst a specification in this aforementioned specification pattern are wait-free. Proving wait-freedom requires us to show that all calls made to operations by any arbitrary client terminate. Thus, as part of formally proving wait-freedom, *i.e.*, as part of the proof of 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 higher-order functions such as the 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 Author: Please fill in 1 or more \ccsdesc macro

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

Digital Object Identifier 10.4230/LIPIcs...

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

```
(* 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

given moment, and forcing others to wait for its completion. However, in many settings, *e.g.*, in operating system kernels [5], 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 acting on the database, at least one of them must run to completion. Yet, with lock-freedom does not guarantee that there is no starvation. That is, given a specific call to a lock-free data structure's operation, we cannot guarantee that that call must terminate. To avoid starvation, one must use so-called *wait-free* [17] 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 whether or not, or how many other operations of the same data structure are run concurrently with it.

We illustrate the differences between blocking, lock-free, and wait-free concurrency using and 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. 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 starvation behavior of this implementation of the concurrent counter data structure depends on whether the lock used is a fair lock or not. 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. Note that if the CAS operation fails, that means that the value of the counter (stored in memory location `l`) must have changed since it was read on the line above the CAS operation. This in turn means that another increment operation must have succeeded in another thread. Thus, this implementation is lock-free. 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 (of type unit to unit).

Of course, in this contrived example of a simple concurrent counter (Figure 1c) we can relegate the problem of making the data structure wait-free to the hardware using the FAA

```
(* The function loop should not make any
   recursive calls. *)
let simple_op () =
  (* Could be allocated at address 0x42. *)
  let b = ref true in
  let rec loop () =
    if !b then
      ()
    else
      loop ()
  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

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. Indeed, to ensure wait-freedom, operations 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 [26] versus the lock-free queue of Michael and Scott [30]). 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 [9, 28] 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 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 [28]. 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 [10]). Another approach to ensuring safety of clients is by requiring them to be verified [9, 32]. In addition to not being applicable in our setting (explained below), 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 [25] 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, for all intents and purposes codify 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 is not applicable. The adequacy applies to closed programs, *i.e.*, not a single data structure. Thus, the adequacy theorem must be applied to the closed program, *i.e.*, the data structure and its client. Moreover, the adequacy theorem states that when a program is proven correct in Lawyer, then the entire program, meaning all threads must terminate. Hence, in this work we state and prove a separate and novel adequacy theorem specifically for wait-freedom. Secondly, the Lawyer logic allows only composition of two programs as long as they are both verified in the Lawyer logic. Thus, if we were to only restrict ourselves to verified clients, we would also be restricted to only considering terminating clients. On the other hand, as we discussed above wait-freedom only makes sense when the client is 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*. 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, 13] have used logical relations models.

We use the approach outlined above for proving wait-freedom to prove wait-freedom of 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 [21]. 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 reasoning about such examples formally requires us to introduce a notion of what we call *restricted* wait-freedom, and to slightly adjust our approach to verification of wait-freedom.

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 notion of wait-freedom in the Lawyer logic
- A solution for verifying reasoning about arbitrary client programs based on combining 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

The rest of the paper is organized as follows. We formally define wait-freedom in Section 2 where we 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 prove such specification for a simple example program to illustrate our approach. In Section 4 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 5. Finally, we discuss a limitations of our solution in Section 6, compare it with related work in Section 7, and conclude in Section 8.

2 Overview

In this section, we formally define the notion of wait-freedom as a property 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 will be using 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 will use this notation for the trace configuration lookup which we define 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.

2.1 Definition of wait-freedom for our language

We consider the programs written in the language $\lambda_{\text{ref}}^{\text{conc}}$— an ML-like higher-order language with general references and concurrency. An excerpt of its semantics shown in Figure 3, whereas the full semantics is provided in the Rocq development. A less common feature of $\lambda_{\text{ref}}^{\text{conc}}$ — `to_int` primitive — is explained further in Section 4.2. The $\mathit{fill}\,(K : \mathit{ECtx})\,(e : \mathit{Expr})$ operation fills the hole $\bullet$ in the context K with the expression e; we avoid the more common $K[e]$ notation, as it is used for lookup operations in this paper.

We define wait-freedom as a property of the execution traces of $\lambda_{\text{ref}}^{\text{conc}}$. Intuitively, an operation $\mathsf{op} : \mathit{Val}$ is wait-free if, for any execution trace, any "call" to op is followed by a "return" later in this trace. A "call" in $\lambda_{\text{ref}}^{\text{conc}}$ 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, identified with a thread identifier and an evaluation context.

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

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

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

To define wait-freedom, we consider execution traces of $\lambda_{\text{ref}}^{\text{conc}}$. Before proceeding, we introduce a number of general notations for traces. Below, $\mathit{tr}[i]$ and $\mathit{tr}\langle i \rangle$ denote correspondingly

Syntax

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

Pure reductions

$$
(\mathtt{rec}\, f\ (x) := e)\ v \overset{\text{pure}}{\rightsquigarrow} e[\mathtt{rec}\, f\ (x) := e/f][v/x] \qquad \mathtt{if}\ \mathtt{true}\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \overset{\text{pure}}{\rightsquigarrow} e_1
$$

$$\vdots$$

Per-thread reductions

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

$$\vdots$$

Configuration reductions

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

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

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

the ith state and label of a trace tr. In particular, for an execution trace etr of $\lambda_{\mathsf{ref}}^{\mathsf{conc}}$ and any index i of it, $tr[i]$ is the configuration (a pair of a thread pool and a heap, according to Figure 3) before the ith step, and $etr\langle i\rangle$ is the identifier of the thread taking the ith step. 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.

We'd like to state that every call must return. Of course, it 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* etr *and thread identifier* τ*, we say that all* τ*'s calls to* op *in* etr *eventually return iff*

$$
\mathit{alwaysReturnsInTrace}(\mathsf{op}, etr, \tau) \triangleq \forall (E : \mathit{ECtx}), i.\ \mathit{Call}(etr[i].1, \tau, E, \mathsf{op}) \implies \mathit{schedUntilRet}(etr, \tau, E, i) \implies \exists j \geq i.\ \mathit{Ret}(etr[j].1, \tau, E)
$$

where

$$
\mathit{schedUntilRet}(etr, \tau, E, i) \triangleq \forall j \geq i.\ j \in \mathsf{dom}\ etr \implies \neg(\exists k \in [i..j].\ \mathit{Ret}(etr[k].1, \tau, E)) \implies \exists p \geq j.\ etr\langle p\rangle = \tau,
$$

Finally, we restrict the starting configuration of traces under consideration:

- The program of every thread in the starting configuration 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*). An example from Figure 2 illustrates why it is necessary to restrict random memory access by the client program.
- 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 starting configurations.

As a result, we define wait-freedom as follows.

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

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

For example, in Section 4 we show that a FAA-based `incr` operation of counter (Figure 1c) is wait-free if the location l of the counter is initialized with an integer. In terms of Definition 3, it is denoted as

$$\mathit{waitFree}_{\mathit{hasInt}(l)}(\texttt{incr}\ l), \quad \text{where} \quad \mathit{hasInt}(l) \triangleq \{c \mid c.2[l] \in \mathbb{Z}\}$$

We note that, while Definition 3 captures the behavior of simple wait-free operations, more complicated algorithms only satisfy weaker forms of that property. We will see examples of it in Section 4.1 and Section 4.2.

2.2 Proving wait-freedom with Lawyer logic

The notion of wait-freedom as defined in Definition 3 is formulated 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 [25] and Trillium [35] frameworks. We explain the details of Lawyer 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 allowed starting configurations $C \subseteq Config$, we define the Lawyer specification of* op*'s wait-freedom as*

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

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

Our adequacy theorem states that the definition 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 operations is more elaborate. Therefore, for such operations we prove slightly different versions of Definition 4.

We show how the specifications from Definition 4 are proven for simple programs in Section 3. Then, in Section 4 we show how our approach can be adapted to verify programs that satisfy weaker versions of Definition 3. Finally, we discuss the key insights behind the proof of Theorem 5 in Section 5.

3 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 the wrapper $\mathtt{incr}\,l$ over the wait-free implementation of counter increment from Figure 1c. Specifically, we verify $\mathsf{WaitFreeSpec}_{\mathit{hasInt}(l)}(\mathtt{incr}\,l)$ in the Lawyer logic for any location l. Doing so allows us to use Theorem 5 to conclude $\mathit{waitFree}_{\mathit{hasInt}(l)}(\mathtt{incr}\,l)$ holds at the meta-level. We will introduce the details of Iris and the Lawyer logic as they are needed.

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

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

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:

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

Iris is a separation logic and thus Iris propositions assert ownership of *resources* representing both the state of the program, and so-called ghost resources. For example, ownership of a single heap location is expressed with a *points-to* proposition, written $\ell \mapsto v$, with the usual intuitive meaning: it asserts both exclusive ownership of the memory location ℓ, and the fact that it currently stores the value v. To assume ownership of the entire heap, we take the *separating conjunction* of points-to propositions for all locations in the heap. The separating conjunction $P * Q$ asserts ownership of resources which can be split into two *disjoint* parts, one satisfying P and the other Q. The notion of being disjoint depends on particular resource kind. For points-to propositions, $l_1 \mapsto v_1 * l_2 \mapsto v_2$ means that $l_1 \neq l_2$; hence the exclusivity of ownership of the memory location. Thus, given the ownership of the entire heap, we have to prove the specifications $\mathsf{NoInfExec}(\mathtt{incr}\,l)$ and $\mathsf{PresModInv}(\mathtt{incr}\,l)$. However, proving both will rely on the fact that l always stores some number. Such a fact can be expressed through Iris *invariants* as follows:

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

An invariant proposition $\boxed{P}$ means that P always holds throughout program's execution. While verifying a single program step, one can assume that P holds by *opening* the invariant, but it must be *closed* again right after taking that step of computation by reestablishing that P holds again. Importantly, invariants (but not their content in general) are duplicable. That is, $\boxed{P} \vdash \boxed{P} * \boxed{P}$. For concurrent operations such as $\mathtt{incr}\,l$ that we consider, it means that every thread running an operation can be given a knowledge about the invariant.

Note that the definition of WaitFreeSpec features a so-called *viewshift*, written $\Rrightarrow$. Viewshifts are similar to logical implication, but additionally allow give access invariants and resources. That is, to prove $P \Rrightarrow Q$, in addition to assuming P, we can open/close invariants and update resources so as to establish Q. Thus, given ownership of the entire heap, and the fact that l currently points to some integer, we establish cntInv by providing $l \mapsto k$ (ignoring the rest of the heap). From that point on we can assume that cntInv holds while proving the specifications.

The first specification we need to prove, NoInfExec(`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[1]:

$$\mathsf{NoInfExec}(\mathsf{op}) \triangleq$$

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

The number $\mathbf{F}$ is (an over-approximation of) the upper bound on the execution time of op. The actual specification for op is defined as a Lawyer *Hoare triple*. In general, a triple $\{P\}\ e\ \{\Phi\}^\tau$ means that assuming that the precondition P holds, the program e without getting stuck (*i.e.*, it does not crash) when run under the thread τ, and that whenever e reduces to a value v, the postcondition $\Phi(v)$ holds. The Hoare triple also enforces that throughout the execution of the program all invariants are preserved. Moreover, Lawyer Hoare triples additionally enforce that every single step of the program e is refined by a transition in the so-called Obligations Model of Lawyer—a transition system that Lawyer instantiates Trillium with. As a result, given a Lawyer Hoare triple for a program e implies that any execution trace of e refines some trace of the Obligations Model. As all traces of the Obligation Model are finite, this refinement allows Lawyer to establish termination of verified programs.

The precondition and postcondition of the triple NoInfExec(op) above consist of Lawyer logic resources. First, the precondition can be read as an assignment of $\mathbf{F}$ barrels of *fuel* to the thread τ, with an indirection via the *phase* π that marks the barrels, and is assigned to τ. 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 is a logical representation of 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. As an aside we mention that in Lawyer [31], both the fuel and the phase of resources have one extra parameter each. Here, 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.

Later on in the paper, in Section 6, we will argue why specifically the specific specification pattern of NoInfExec(op), as well as certain restrictions on the operation's invariants, captures the nature of wait-freedom in the Lawyer logic.

To prove NoInfExec(`incr` l), we take $\mathbf{F}$ to be 3, this requires us to show that for any τ, π, and a we have the following:

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

The triple above essentially states that we can spend up to 3 barrels of fuel to complete the execution of op a. To prove this triple, we use the rules of the Lawyer logic [31]. The Lawyer

[1] 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].

logic provides rules that cover updates to both the physical state of execution (*i.e.*, the heap) ,and the state of the Obligations Model. For simplicity, we only show a number of derived rules that are necessary for our specific example.[2] An execution of $\texttt{incr}\, l\, a$ consists of two beta-reductions and one FAA step. For the former, we use the following rule:

Lawyer-beta-fuel
$$\{\mathsf{ph}_\circ \tau\, \pi * P\}\, e[v/x]\, \{\mathsf{ph}_\circ \tau\, \pi\}^\tau \vdash \{\mathsf{bar}_\circ \pi * \mathsf{ph}_\circ \tau\, \pi * P\}\, (\lambda x.\ e)\, v\, \{\mathsf{ph}_\circ \tau\, \pi\}^\tau$$

The rule above says that to verify a beta-reduction, user must spend a barrel of fuel and preserve the phase resource. After the beta-reduction step the user can continue verifying the reduct with the remaining resources. Thus, we apply this rule twice by choosing P to be $2{\cdot}\mathsf{bar}_\circ \pi$ and $\mathsf{bar}_\circ \pi$, respectively. After these two beta-reduction steps, we are left to prove the following:

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

The fetch-and-add primitive can only proceed when the location it accesses contains an integer, which is the case for our example as evidenced by our invariant. Thus, to verify the FAA step we need to open the invariant, get the current value under location l, increment it, and close the invariant again. Moreover, as with the previous steps, we need to spend fuel for this step as well. All of that is captured by the following rule:

Lawyer-FAA-fuel-inv
$$\frac{P \mathrel{-\!\!*} \exists m : \mathbb{Z}.\, \ell \mapsto m * (\ell \mapsto (m+n) \mathrel{-\!\!*} P)}{\boxed{P} \vdash \{\mathsf{bar}_\circ \pi * \mathsf{ph}_\circ \tau\, \pi\}\, \textbf{FAA}(\ell, n)\, \{\_.\, \mathsf{ph}_\circ \tau\, \pi\}^\tau}$$

The *(magic) wand* connective of separation logic, written $\mathrel{-\!\!*}$, is a separating implication: $P \mathrel{-\!\!*} Q$ holds for a resource such that when added to any resource satisfying P, once can satisfy Q. Hence, the premise of the rule Lawyer-FAA-fuel-inv says that the content of invariant should contain a points-to resource for the location, and that the invariant could be reestablished after the location has been updated. This is trivially satisfied by cntInv. The expression $\textbf{FAA}(l, 1)$ reduces to the value stored under l in the heap before the update takes place. This value is ignored by our postcondition which only requires the phase resource. Thus, after using the rule Lawyer-FAA-fuel-inv the proof of $\mathsf{NoInfExec}(\texttt{incr}\, l)$ is finished.

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

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

The overall intuitive meaning of this triple is similar to one described above: if the precondition holds, then an execution of $\mathsf{op}\, a$ preserves invariants, and if it reduces to a value, then the postcondition holds for that value. However, there are two major differences. First, the execution is allowed to get stuck (denoted by $\lightning$ superscript). Second, this triple does not establish a refinement between the program and the Obligation model and thus its precondition and postcondition, as well as invariants it relies on, cannot mention any Lawyer resources.[3] We use the noMod subscript to denote that the triple does not deal with

[2] The rules that we present below are derived from the rules of Lawyer logic which are more general for more flexibility. However, that level of generality and flexibility is not necessary for the results presented in this paper, and thus presenting them would only muddle the presentation.

[3] 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

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

(b) Representation of lists in $\lambda_{\text{ref}}^{\text{conc}}$

Figure 4 Higher-order list mapping function

refinement to a model. As we will explain in Section 5.2, 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 the details of what it means to be safe value and the motivation as to why it is necessary to restrict only to safe values in Section 5.2.

The proof of $\mathsf{PresModInv}(\texttt{incr}\, l)$ proceeds similarly to one above. Since the input argument a is not used, we ignore the $\mathsf{RS}_{\mathbf{V}}(a)$ resource. The rules used to verify beta-reductions and FAA can be obtained by simply removing Lawyer resources from the corresponding rules above and adjusting the kind of Hoare triple. Similarly to the proof above, we use cntInv to verify the FAA step. The main difference 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, as we will see in Section 5.2 all integers are safe, *i.e.*, $\forall n : \mathbb{Z}.\ \vdash \mathsf{RS}_{\mathbf{V}}(n)$.

4 Case studies

As we discussed in the Introduction, while the simple definition of wait-freedom that we worked with in Sections 2 and 3 works for many simple programs, for more involved programs, this definition needs to be adjusted. In this section, we motivate and explain two independent extensions to our approach to verification of wait-freedom. One where we allow to verify (higher-order) functions that potentially get stuck, and one where the data structure restricts the number of operations that can run concurrently at a given time. Throughout this section we will use the following helper definitions:

$$\mathbf{None} \triangleq \mathbf{inl}\,() \qquad \mathbf{Some}\, v \triangleq \mathbf{inr}\, v$$

and similarly define a corresponding notation for in pattern matching where we simply write `match e with None => e1 | Some v => e2 end`.

4.1 Modular verification of higher-order wait-free functions

Consider the higher-order `list_map` function in Figure 4a. It applies the given function `f` to all elements of the list l. In our language, lists are values satisfying the recursive predicate isList defined in Figure 4b.

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 behavior 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* τ*, we say that all* τ*'s calls to* op *in etr eventually return or get stuck, written* $\mathit{alwaysReturnsInTrace}^{\lightning}(\mathtt{op}, \mathit{etr}, \tau)$*, iff*

$$\mathit{alwaysReturnsInTrace}^{\lightning}(\mathtt{op}, \mathit{etr}, \tau) \triangleq$$
$$\forall \mathcal{E}, i.\ \mathit{Call}(\mathit{etr}[i].1, \tau, E, \mathtt{op}) \land \mathit{schedUntilRet}(\mathit{etr}, \tau, E, i) \implies$$
$$\exists j \geq i.\ \mathit{Ret}(\mathit{etr}[j].1, \tau, E) \lor \mathit{stuck}(\mathit{etr}[j], \tau)$$

where stuckness is defined as follows:

$$\mathit{stuck}(c, \tau) \triangleq \neg \exists c'.\ c \xrightarrow{\tau} c'$$

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

Our approach to proving wait-freedom supports proving verifying possibly-stuck wait-freedom with minimal changes compared to the ordinary wait-freedom we had discussed in earlier section. 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 in turn, similarly to definitions above, is a weakening of WaitFreeSpec that allows the operation to possibly get stuck.

Now, we can apply the approach outlined in Sections 2 and 3 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`:

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

As in Section 3, 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 3, with the addition of rules to verify stuck computations, *e.g.*, the following Hoare triple for projecting of the unit value: $\{\mathsf{True}\}^{\lightning}\ \mathtt{fst}\ ()\ \{P\}^{\tau}$.

Given the (list-map-HO-spec) spec above, together with the fact that ordinary Hoare triples imply possibly-stuck Hoare triples, we can reuse already the 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 3, we obtain a proof of $\mathsf{WaitFreeSpec}^{\lightning}_{\mathit{hasInt}(l)}(\mathtt{list\_map\ (incr}\ l))$ for any location l, thus establishing $\mathit{waitFree}^{\lightning}_{\mathit{hasInt}(l)}(\mathtt{list\_map\ (incr}\ l))$.

4.2 Restricted wait-freedom

Many practical implementations of wait-free algorithms [21, 27, 36, 14] 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 5a. 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 5a limits the concurrency to just two threads: an enqueuer thread that can both enqueue elements and read the queue head, and a dequeuer thread that can both dequeue elements and read the queue head.

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

```
(* 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 [21] expressed in $\lambda_{\mathsf{ref}}^{\mathsf{conc}}$

```
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 (dequeue ())
  else
    None
```

(b) Wrappers to be verified

Figure 5 Single-enqueuer, single-dequeuer queue

To support verifying such algorithms, which we call *restricted wait-free* requires multiple changes to our approach to verification of wait-freedom as we have described so far. First of all, we define two wrapper functions `enqueuer` and `dequeuer` (see Figure 5b) 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 6, 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 $\lambda_{\mathsf{ref}}^{\mathsf{conc}}$ 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 can be stated as a variant of Definition 3 as follows, with changes highlighted:[4]

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

▶ **Definition 7** (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*

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

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^{th} thread is a client of the i^{th} method in ops. Importantly, none of threads threads should fork any other threads—otherwise the restriction on the number of concurrent operations might be violated. The *alwaysReturnsInTrace* conclusion states that every call to one of the operations in any of the threads must always return.

For example, restricted wait-freedom of the queue algorithm in Figure 5b is formally stated using Definition 7 by taking $\texttt{ops} \triangleq [\texttt{enqueuer}, \texttt{dequeuer}]$. For more complicated n-enqueuer, single-dequeuer algorithm (also presented in [21]) one would need 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 allowed starting configurations $C \subseteq Config$, we define the Lawyer specification of* $\texttt{ops} : List(Val)$ *restricted wait-freedom as follows, for some* $\mathbf{F} : \mathbb{N}$*:*

$$\mathsf{WaitFreeRestrSpec}_C(\texttt{ops}) \triangleq \forall c \in C.\ \mathsf{heap\_repr}(c) \Rrightarrow \mathsf{mtoks}(\texttt{ops}) *$$
$$\mathop{\Large *}_{\texttt{op} \in \texttt{ops}} \left(\forall \tau, \pi, a.\ \{\mathbf{F} \cdot \mathsf{bar}_\circ \pi * \mathsf{ph}_\circ \tau\, \pi * \mathsf{mtok}(\texttt{op})\}\ \texttt{op}\ a\ \{\_.\ \mathsf{ph}_\circ \tau\, \pi * \mathsf{mtok}(\texttt{op})\}^\tau\right) *$$
$$\mathop{\Large *}_{\texttt{op} \in \texttt{ops}} \left(\forall \tau, a.\ \{\mathsf{mtok}(\texttt{op})\}^{\natural}_{\mathsf{noMod}}\ \texttt{op}\ a\ \{v.\ isGroundVal(v) * \mathsf{mtok}(\texttt{op})\}^\tau\right)$$

where

$$\mathsf{mtoks}(l : List(Val)) \triangleq \mathsf{toks}_{\texttt{ops}}(l) \quad \text{and} \quad \mathsf{mtok}(m : Val) \triangleq \mathsf{mtoks}([m])$$

In the definition above, $isGroundVal(v : Val)$ 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 6. The proposition $\mathsf{mtoks}(l)$ (an abbreviation for more general $\mathsf{toks}_{\texttt{ops}}(l)$) is a proposition which asserts ownership of the list l of *method tokens*. The list of tokens can be split into individual tokens, and most importantly, the total amount of tokens is determined upon creation and cannot be increased. Thus, requiring ownership of $\mathsf{mtok}(\texttt{op})$ for calling an operation op allows us to restrict the number of calls to op in the program logic. In logic, this is expressed by the following laws (where all parameters of toks are lists of values and $\mathsf{count}(m, l)$ counts occurrences of m in list l):

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

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

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

We use Theorem 9 to establish $\mathit{waitFreeRestr}_{C_Q}([\texttt{enqueuer}, \texttt{dequeuer}])$ (where C_Q is some set of allowed initial states for the queue in Figure 5a) by verifying the wrappers in Figure 5b. 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.

5 Proving the wait-freedom adequacy theorem

As the example in Section 3 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 4. Thus, the non-trivial parts of the proof might only concern safety of the operation (as we mentioned in Section 4.2).

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.

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. For example, there are no rules that allow generating fuel indefinitely long to be spent in an infinite execution.

We address these challenges as follows. First, in Section 5.1 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 C1. Then, in Section 5.2 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 C2. Finally, in Section 5.3 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 C3.

Finally, we note that Theorem 5 and its variants for possibly-stuck wait-freedom from Section 4.1 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.

5.1 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 op and some set of starting configurations $C \subseteq \mathit{Config}$ we assume that an execution trace etr, evaluation context E, thread τ and trace index i such that:

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

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

To prove that 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 C2 and Challenge C3, the Lawyer logic can only support terminating programs, whereas we already concluded above that 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 5.3 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 Trillium (and Lawyer) 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" which we describe in more detail in Section 5.3. 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 ξ on finite traces of $\lambda_{\mathsf{ref}}^{\mathsf{conc}}$ and the model. Then, consider* $\mathsf{TI} : \mathit{FinTrace}(\lambda_{\mathsf{ref}}^{\mathsf{conc}}) \to \mathit{FinTrace}(\mathcal{M}) \to \mathit{iProp}$ *— 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"* $\mathsf{PR}_F : \mathit{FinTrace}(\lambda_{\mathsf{ref}}^{\mathsf{conc}}) \to \mathit{iProp}$ *that satisfies a number of laws described in appendix[2].*

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

$$\vdash \Rrightarrow \mathsf{TI}(etr[0], \mathsf{m}) * \mathsf{AlwaysHolds}(\xi, etr[0], \mathsf{m}) * \mathsf{PR}_F(etr[0]) \; (* \textit{ treating } etr[0] \textit{ as a singleton trace } *)$$

and $F(etr')$ *holds for any finite prefix* etr' *of* etr*, 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 AlwaysHolds and $\hat{\xi}$; the latter is essentially the refinement we're 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 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 5.3; 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 5.3. Thus, using the Trillium's adequacy theorem we prove that *etr* is finite, and thus derive a contradiction.

5.2 Logical relation for ensuring "robust safety" of a wait-free operation

As the Challenge C2 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 C3 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 smaller 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 5.3, is showing that any execution of a client can be represented in the Obligations Model of Lawyer.

To prove robust safety, we employ the well-known idea of logical relations [34, 13]. 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 3 we described a PresModInv Hoare triple that states a property similar to $\mathsf{RS}_{\mathbf{E}}(\cdot)$. Indeed, expression relation is defined similarly to PresModInv (we explain $\mathsf{RS}_{\mathbf{V}}(v)$ below):

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

To understand how the expression relation 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 a *value relation* $\mathsf{RS}_{\mathbf{V}}(v)$ defined as follows:[5]

▶ **Definition 11** (Value relation).

$$\mathsf{RS}_{\mathbf{V}}(()) \triangleq \mathsf{RS}_{\mathbf{V}}(b : \mathbb{B}) \triangleq \mathsf{RS}_{\mathbf{V}}(n : \mathbb{Z}) \triangleq \mathsf{True}$$

$$\mathsf{RS}_{\mathbf{V}}(\texttt{inl}\ v) \triangleq \mathsf{RS}_{\mathbf{V}}(\texttt{inr}\ v) \triangleq \mathsf{RS}_{\mathbf{V}}(v)$$

$$\mathsf{RS}_{\mathbf{V}}((v_1, v_2)) \triangleq \mathsf{RS}_{\mathbf{V}}(v_1) * \mathsf{RS}_{\mathbf{V}}(v_2)$$

$$\mathsf{RS}_{\mathbf{V}}(\ell : \mathit{Loc}) \triangleq \boxed{(\exists v : \mathit{Val}.\ \ell \mapsto v * \mathsf{RS}_{\mathbf{V}}(v)) \vee \mathsf{freed}(\ell)}$$

$$\mathsf{RS}_{\mathbf{V}}(\texttt{rec}\ f\ x := e) \triangleq \forall \tau, v.\ \{\mathsf{RS}_{\mathbf{V}}(v)\}^{\not\downarrow}_{\mathsf{noMod}}\ (\texttt{rec}\ f\ x := e)v\ \{r.\ \mathsf{RS}_{\mathbf{V}}(r)\}^{\tau}$$

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 [23] 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[6], showing that evaluating its sub-expressions produce values that can be safely used afterwards.

However, for an e that actually calls a wait-free operation `op`, it 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 5a 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 $\mathsf{PresModInv}(\texttt{op})$ 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 $\mathsf{PresModInv}(\texttt{op})$ 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:

▶ **Theorem 12** (Fundamental theorem). *Let e be an expression that does not contain any Loc literals, or any pointer arithmetic, and whose free variables are $\vec{xs}$. Given any list of values $\vec{vs}$ such that $|\vec{vs}| = |\vec{xs}|$. Then the following holds:*

$$\left(\mathop{\Large *}_{v \in \vec{vs}} \mathsf{RS}_{\mathbf{V}}(v)\right) \vdash \mathsf{RS}_{\mathbf{E}}(e[\vec{vs}/\vec{xs}])$$

A client of the operation `op` 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`,

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

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

we apply Theorem 12 and provide $\mathsf{PresModInv}(\mathsf{op})$ to satisfy its premise. As a result, 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 does not contain any Loc literals, or any pointer arithmetic. Assume that $\mathsf{WaitFreeSpec}_C(\mathsf{op})$ holds for some C. Then the following holds:*

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

5.3 Refining an infinite execution to a trace of Obligations Model

In Section 5.1, we have reduced proving wait-freedom to that of termination, which we establish by applying Theorem 10 and refining a given execution to the 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 C3 states, the Lawyer logic that establishes refinement of the program by the Obligations Model was designed to only verify terminating programs. Thus, in this section we briefly describe the solution that allows us to repurpose the Lawyer logic for verifying non-terminating programs. Due to lack of space we refer the reader to our Rocq formalization [1] for the details.

The central part in establishing refinement by verifying a program is the progress resource used by Theorem 10. 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. By providing the progress resource for a singleton trace $\mathsf{PR}_F(c)$, one essentially proves 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 [2]. 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 construction of 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 5.1. Namely, that our progress resource is defined as $F(etr) \triangleq \mathit{infCallPrefix}(E, \tau, i, etr)$, where E, τ and i are fixed during the reduction. We refer the reader to Rocq development [1] for its formal definition. Intuitively, this predicate means that etr is a possible finite prefix of some infinite trace in which thread τ 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 τ 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 τ 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, τ 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 5.1 by contradiction—under normal circumstances, a thread with an obligation will eventually fulfill it and thus disable fuel generation.

6 Discussion and limitations

6.1 Lawyer specifications of non-blocking properties

In Section 3, we claimed that the specification NoInfExec(op) captures that all calls to op must terminate. We, furthermore claimed that PresModInv(op) implies that op does not rely on any invariants that involve Lawyer resources. Thus, according to Liang and Feng [28], these conditions together imply that op is wait-free as it must never block itself, nor can it delay other threads. Below we will explain how these specifications prohibit blocking and delaying, and discuss the changes needed in our approach in order to support other non-blocking properties.

To understand why NoInfExec(op) prohibits blocking, we need to explain what this specification *does not* include. Remember that it is defined as follows, for some fixed $\mathbf{F} : \mathbb{N}$:

$$\forall \tau : \mathsf{Thread}, \pi : \mathsf{Phase}, a : \mathit{Val}.\ \{\mathbf{F} \cdot \mathsf{bar}_\circ \pi * \mathsf{ph}_\circ \tau\, \pi\}\ \mathsf{op}\ a\ \{\_.\ \mathsf{ph}_\circ \tau\, \pi\}^\tau$$

Neither its precondition- nor postcondition mention any obligation resource $\mathsf{obls}\,\tau\,\mathcal{O}$ of Lawyer. This resource indicates that the thread τ holds a set $\mathcal{O}$ of *obligations* that it must fulfill, *e.g.*, setting a shared flag that other threads can wait for. In Lawyer, a thread having an obligation allows the other threads to busy-wait for this obligation to be fulfilled. In practice, waiting of another thread's obligation means that the waiting thread can *generate* fuel to be spent during its busy-waiting. To prevent circular waiting, a thread can only generate fuel by presenting its own set of obligations, and crucially proving that all their obligations are "ordered after" the obligation (of the other thread) that it waits for.[7] Thus, NoInfExec(op) not including obligations resources means that it is non-blocking, because it cannot generate fuel by waiting for other threads' obligations, and thus it can only spend the fuel that is explicitly provided to it in NoInfExec(op).

A delay in a lock-free algorithm is a situation when the successful completion of one operation makes another operation running concurrently with it to fail and have to retry. An example of it can be seen in Figure 1b where a successful **CAS** in one thread can cause

[7] For more details of what this "ordered after" relation is see the Lawyer paper. We only emphasize here that this order is well-founded.

another thread to fail its own **CAS** which forces it attempt to increment again by making a recursive call. To prohibit delays, our approach disables the fuel sharing mechanism of Lawyer. This mechanism, introduced by [19], requires a thread that delays others to provide them with extra fuel: since the delayed thread will retry its operation, it will need more fuel. In Lawyer (see [31, Delaying example]), the fuel is shared via an invariant that is accessible by both the delaying thread, and the delayed thread. To disable fuel sharing, we simply require that invariants of the wait-free operation do not contain any Lawyer resources.[8]

Besides wait-freedom, there are two other non-blocking properties: obstruction-freedom [16] and lock-freedom. The former only requires an operation to progress if it is delayed by other operations a finite number of times. When this condition is violated, an obstruction-free operation might block. Thus, we believe that adapting our approach to verifying obstruction-freedom could be achieved by adding obligation resources to the specification. For lock-freedom, one could follow the approach of Lawyer and enable fuel sharing by allowing fuel sharing, *i.e.* by lifting the aforementioned restriction on invariants. However, the soundness of fuel sharing in Lawyer is relies on the "phases" mechanism [19], which only allows to share fuel among the threads that have the same ancestor in the fork tree. The definition of lock-freedom does not place any restrictions on threads' ancestry, and therefore this restriction too must be lifted from the fuel sharing mechanism.

6.2 Restricting the content of wait-free data structures

When verifying the queue implementation in Section 4.2, we introduced a seemingly unnecessary constraint. Namely, we defined `enqueuer` wrapper in Figure 5b in a way that only allows the integer values to be enqueued. Note that the original implementation in Figure 5a never inspects the enqueued values and thus does not rely on this constraint. However, 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 4.2, 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 5a using Theorem 13 requires proving the PresModInv(`dequeuer`) specification, which is the lambda-expression case of Definition 11. (For technical reasons, Definition 8 requires a stronger specification from which PresModInv(`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

[8] More formally, we require that the invariant can be defined in non-Lawyer instantiations of Trillium.

non-terminating call to `enqueuer` with some argument a. Internally, the adequacy theorem uses (the token-based version of) $\mathsf{NoInfExec}(\texttt{enqueuer})$ to verify the call $\texttt{enqueuer}\,a$. 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.

6.3 Proving linearizability of wait-free data structures

An important criteria for the correctness of a concurrent data structure is *linearizability* [18]. 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 [21, 27, 36, 14]. 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 [20, 8, 25, 4]. Birkedal *et al.* [4] have formally shown that linearizability is internalized by specifications given in terms of the so-called *logically atomic Hoare triples* [25]. 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 5a. 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 [24]. 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.

6.4 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 [9] 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 $\mathsf{PresModInv}(\texttt{op}')$ for every lock-free operation $\texttt{op}'$ running concurrently with `op`.

7 Related work

All prior solutions verify wait-freedom for first-order languages. These solutions all guarantee wait-freedom under the assumption, in one way or another, that the client of a wait-free operation is safe, *i.e.* that the client does not violate the data structure's internal invariants. In our case, 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 5.2. Moreover, most of the prior solutions only support unrestricted wait-freedom, lack case studies, and are not mechanized.

7.1 Program logics for wait-freedom

Total TaDA The program logic of [9] 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 both while loops and recursive functions by choosing a decreasing measure of termination. Moreover, their specifications are also capable of expressing the linearizability, which we do not consider.

Da Rocha Pinto *et al.* [9] demonstrate their approach 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 [22] 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 [8] logic. Contrary to that, our approach supports arbitrary client programs (up to assumptions of Theorem 12). Finally, Total Tada is not mechanized.

Lili The approach of [28] 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 [29] have any case studies on wait-freedom verification.

Automatic tool for verifying non-blocking algorithms The work of [15] 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.* [15] 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.* [15] 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 [15] 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 5a 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 [27]. 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 [3] 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.* [15], 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 [27], which is in turn an optimization of the queue by Kogan and Petrank [26]. In the implementation by [27], 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.

7.2 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 [17] 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 the universal constructions has a mechanized proof of wait-freedom.

Therefore, it is interesting to consider 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 that is similar to how we verified possibly-stuck wait-freedom of `list_map (incr l)` in Section 4.1. Therefore, we believe that our approach can be applied to universal constructions as well.

Examples of modern wait-free universal constructions include P-Sim [11] (along with its extensions such as [12]) and CX [7]. 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.

8 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 spec 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 [6]. 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 Anonymous. Supplementary rocq development for the wait-freedom paper, 2026.

2 Anonymous. Technical appendix for the wait-freedom paper, 2026.

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

4 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), aug 2021. doi:10.1145/3473586.

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

6 Hagit Brit and Shlomo Moran. Wait-freedom vs. bounded wait-freedom in public data structures (extended abstract). In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '94, page 52–60, New York, NY, USA, 1994. Association for Computing Machinery. doi:10.1145/197917.197950.

7 Andreia Correia, Pedro Ramalhete, and Pascal Felber. A wait-free universal construction for large objects. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '20, page 102–116, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3332466.3374523.

8 Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. Tada: A logic for time and data abstraction. In Richard Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, pages 207–231, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

9 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.

10 Derek Dreyer, Georg Neis, and Lars Birkedal. The impact of higher-order state and control effects on local relational reasoning. *SIGPLAN Not.*, 45(9):143–156, September 2010. doi:10.1145/1932681.1863566.

11 Panagiota Fatourou and Nikolaos D. Kallimanis. A highly-efficient wait-free universal construction. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, page 325–334, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1989493.1989549.

12 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. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.OPODIS.2019.18, doi:10.4230/LIPIcs.OPODIS.2019.18.

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

14 Seep Goel, Pooja Aggarwal, and Smruti R. Sarangi. A wait-free stack. In Nikolaj Bjørner, Sanjiva Prasad, and Laxmi Parida, editors, *Distributed Computing and Internet Technology*, pages 43–55, Cham, 2016. Springer International Publishing.

15 Alexey Gotsman, Byron Cook, Matthew Parkinson, and Viktor Vafeiadis. Proving that non-blocking algorithms don't block. *SIGPLAN Not.*, 44(1):16–28, January 2009. doi:10.1145/1594834.1480886.

16 M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: double-ended queues as an example. In *23rd International Conference on Distributed Computing Systems, 2003. Proceedings.*, pages 522–529, 2003. doi:10.1109/ICDCS.2003.1203503.

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

18 Maurice P. 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.

19 Bart Jacobs, Dragan Bosnacki, and Ruurd Kuiper. Modular termination verification of single-threaded and multithreaded programs. *ACM Trans. Program. Lang. Syst.*, 40(3):12:1–12:59, 2018. doi:10.1145/3210258.

20 Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, page 271–282, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1926385.1926417.

21 Prasad Jayanti and Srdjan Petrovic. Logarithmic-time single deleter, multiple inserter wait-free queues and stacks. In Sundar Sarukkai and Sandeep Sen, editors, *FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science*, pages 408–419, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

22 C. 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.

23 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.

24 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), dec 2019. doi:10.1145/3371113.

25 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.

26 Alex Kogan and Erez Petrank. Wait-free queues with multiple enqueuers and dequeuers. *SIGPLAN Not.*, 46(8):223–234, February 2011. doi:10.1145/2038037.1941585.

27 Alex Kogan and Erez Petrank. A methodology for creating fast wait-free data structures. *SIGPLAN Not.*, 47(8):141–150, February 2012. doi:10.1145/2370036.2145835.

28 Hongjin Liang and Xinyu Feng. A program logic for concurrent objects under fair scheduling. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 385–399, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2837614.2837635.

29 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.

30 Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, page 267–275, New York, NY, USA, 1996. Association for Computing Machinery. doi:10.1145/248052.248106.

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, 2026.

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), jan 2024. doi:10.1145/3632851.

36 Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. Wait-free linked-lists. *SIGPLAN Not.*, 47(8):309–310, February 2012. doi:10.1145/2370036.2145869.

37 Viktor Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2008.

38 Martin Vechev and Eran Yahav. Deriving linearizable fine-grained concurrent objects. *SIGPLAN Not.*, 43(6):125–135, June 2008. doi:10.1145/1379022.1375598.