Lawyer: Modular Obligations-Based Liveness Reasoning in Higher-Order Impredicative Concurrent Separation Logic

EGOR NAMAKONOV, Aarhus University, Denmark
JUSTUS FASSE, KU Leuven, Belgium
BART JACOBS, KU Leuven, Belgium
LARS BIRKEDAL, Aarhus University, Denmark
AMIN TIMANY, Aarhus University, Denmark

Higher-order concurrent separation logics, such as Iris, have been tremendously successful in verifying safety properties of concurrent programs. However, state-of-the-art attempts to verify liveness properties in such logics have so far either lacked modularity (the ability to compose specifications of independent modules), or they have been far too complex to mechanize in a proof assistant. In this work, we introduce Lawyer — a mechanized program logic for modular verification of (fair) termination of concurrent programs. Lawyer draws inspiration from state-of-the-art approaches that use obligations for specifying and proving termination. However, unlike these approaches, which incorporate obligations by instrumenting the source code with erasable auxiliary code and state, Lawyer avoids such instrumentations. Instead, Lawyer incorporates obligations into the logic by embedding them into a purely logical labeled transition system that the program is shown to refine — this makes Lawyer far more amenable to mechanization. We demonstrate the expressivity of Lawyer by verifying termination of a range of examples, including modular verification of a client program whose termination relies on correctness of a fair lock library, and (separately) proving that a ticket lock implementation implements that library's interface. To the best of our knowledge, Lawyer is the first mechanized program logic that supports modular higher-order impredicative liveness specifications of program modules. All the results that appear in the paper have been mechanized in the Rocq proof assistant on top of the Iris separation logic framework.

CCS Concepts: • **Theory of computation** → **Logic and verification**; *Higher order logic*; *Hoare logic*; **Separation logic**; Operational semantics; Invariants; *Program verification*; • **Computing methodologies** → Concurrent programming languages; • **Software and its engineering** → *Software verification.*

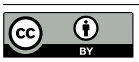
Additional Key Words and Phrases: separation logic, refinement, higher-order logic, concurrency, formal verification, termination, liveness, impredicativity

ACM Reference Format:
Egor Namakonov, Justus Fasse, Bart Jacobs, Lars Birkedal, and Amin Timany. 2026. Lawyer: Modular Obligations-Based Liveness Reasoning in Higher-Order Impredicative Concurrent Separation Logic. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 132 (April 2026), 26 pages. https://doi.org/10.1145/3798240

1 Introduction

An important aspect of program verification frameworks [Appel 2011; Gu et al. 2018; Jacobs et al. 2011; Jung et al. 2015; Lee et al. 2023; Liang and Feng 2016; Sergey et al. 2015] is *modularity*: that different modules of a program can be verified in isolation with respect to their individual

Authors' Contact Information: Egor Namakonov, Aarhus University, Aarhus, Denmark, e.namakonov@cs.au.dk; Justus Fasse, KU Leuven, Leuven, Belgium, justus.fasse@kuleuven.be; Bart Jacobs, KU Leuven, Leuven, Belgium, bart.jacobs@kuleuven.be; Lars Birkedal, Aarhus University, Aarhus, Denmark, birkedal@cs.au.dk; Amin Timany, Aarhus University, Aarhus, Denmark, timany@cs.au.dk.



ACM 2475-1421/2026/4-ART132
https://doi.org/10.1145/3798240

specifications. The benefits of modularity are twofold: (1) it allows us to manage the complexity of the correctness proof by decomposing the problem into smaller, more manageable tasks, and (2) it enables proof reuse where each program module is proven correct once, even though it is used multiple times. Modularity is especially important for the verification of concurrent programs; each thread should be verified in isolation from other threads, lest we would have to explicitly reason directly in terms of different thread interleavings. Modular reasoning about *liveness* properties of concurrent programs is notoriously difficult — here by a liveness property we mean a property that expresses that something good will eventually happen, *e.g.*, the program terminates, or a server eventually responds to client requests. This limitation is mainly due to two reasons. First, the rely-guarantee style reasoning [Jones 1983] used to achieve modularity when reasoning about the safety of concurrent programs is not generally sound for reasoning about liveness properties [Abadi and Lamport 1995]. Second, the step-indexing technique [Appel and McAllester 2001] used to model higher-order concurrent program logics that support highly-modular specifications [Jung et al. 2018] is inherently at odds with liveness reasoning [Hobor et al. 2010; Spies et al. 2021; Timany et al. 2024]. In the rest of this Introduction, we discuss these issues in detail and explain the very recent progress that state-of-the-art research has made to tackle these challenges [Fasse and Jacobs 2024; Timany et al. 2024] before explaining our contributions.

Thread-local modular reasoning about concurrent programs is challenging because threads do not *run* in isolation — rather, they interact, *e.g.*, via shared memory. Thus, to support modular verification of concurrent programs, program verification frameworks must provide a way to express how threads interact with each other. This has traditionally been done in terms of so-called rely and guarantee conditions [Jones 1983] where, for each thread we specify what it expects (relies upon) from other threads and what it guarantees to other threads. Modern program logics use so-called invariants [Jung et al. 2015] — a symmetric form of rely-guarantee: declaring a property P to be an invariant allows all threads to assume that P holds at all times, but at the same time it also forces them to guarantee that P holds whenever they interact with shared memory. To achieve full modularity when reasoning about concurrent programs, the logic used to express and prove the correctness of programs must be higher-order in the sense that any property whatsoever must be allowed to be declared as an invariant. Such a property itself could potentially be defined in terms of invariants — hence, such invariants are often called impredicative invariants [Jung et al. 2018].

To see the need for modularity and impredicative invariants, consider the example in Listing 1. In this example, the `runAsync` function takes a function f, runs it in a separate thread, and in the main thread waits for it to be done, *i.e.*, for the shared optional reference to be set. It then returns the value produced by the call to f in the forked thread. Intuitively, `runAsync` f should satisfy exactly the same properties as f. This can be stated in terms of Hoare-style specifications as follows:

Listing 1. Example program

```
let runAsync f =
  let x = ref None in
  fork (x := Some (f ()));
  (rec loop () =
    match !x with
    | Some v -> v
    | _ -> loop ()) ()
```

$$\forall f, P, \Phi.\ \{P\}\ f\ ()\ \{\Phi\} \implies \{P\}\ \texttt{runAsync}\ f\ \{\Phi\} \qquad \text{(runAsync-spec)}$$

The specification (runAsync-spec) above quantifies over all possible functions f, all propositions P, and all predicates Φ, stating that `runAsync` f satisfies exactly the same Hoare triple as f itself. (The postcondition in our Hoare triples are predicates over values; $\Phi(v)$ is guaranteed to hold for the value v obtained from the computation.) This specification of `runAsync` is reusable in that it can be applied to any function f whatsoever, with any specification (pre- and postcondition) whatsoever. In order to prove `runAsync` correct *modularly*, *i.e.*, showing correctness of the forked thread in isolation from the main thread of `runAsync`, and all other threads that may exist in the system for that matter, we need to establish an invariant which essentially captures that if the value

stored in shared location x is `Some v`, then $\Phi(v)$ should hold, where Φ is an arbitrary predicate picked by the user of the `runAsync` module (and Φ could potentially involve invariants itself).

This expressivity of state-of-the-art program logics [Appel 2011; Jung et al. 2015] comes at a cost. In order to avoid the inherent circularity caused by higher-order features like impredicative invariants, the models of these logics are based on the so-called step-indexing technique. In these logics, proving a property P amounts to proving that, for any $n \in \mathbb{N}$, if we run the program for n steps, the property P holds. While this is sufficient for safety properties (properties that state nothing bad ever happens, *e.g.*, the program does not crash), it inherently precludes liveness reasoning [Hobor et al. 2010]; looking at one finite run at a time, it is impossible to show that something eventually good happens for the continuation of that run. Timany et al. [2024] have, however, recently proposed the Trillium program logic that enables an indirect way of reasoning about liveness properties in the presence of step-indexing. Trillium's program logic allows users to show that programs refine labeled transition system (LTS) models. This refinement relation is set up so that any property, be it safety or liveness, satisfied by the LTS model is also satisfied by the program. Showing that such a strong notion of refinement, which allows transporting both safety and liveness properties, can be carried out in a step-indexed program logic is one of the main contributions of Timany et al. [2024]. However, whereas the program logic of Trillium enjoys all the modularity properties of state-of-the-art program logics in proving the refinement, the Trillium methodology lacks an important aspect of modularity in that one needs to come up with a single LTS model of the entire program that is to be verified.

The Sassy logic, recently developed by Fasse and Jacobs [2024], supports modular reasoning about termination (an important class of liveness properties) of higher-order concurrent programs. The core idea of Sassy is to parameterize blocking modules with auxiliary functions that produce *fuel* when progress is blocked on the client. Via their notion of fuel and other auxiliary state and code, in the sense of Owicki and Gries [1976], the liveness property of termination is reduced to a safety property (of a closely related program).

For sequential programs, it suffices to consume fuel at every step. If the fuel is well-founded, and we can prove that the program cannot exhaust its fuel (get stuck), then the program must eventually terminate. Reinhard and Jacobs [2021a,b] observe that fuel *generation* is sound and supports busy-waiting programs under certain conditions. Intuitively, a program in which threads can wait for one another only terminates if no thread, directly or indirectly, waits for itself. This is ensured by a well-founded order on *obligations* [Boström and Müller 2015; D'Osualdo et al. 2019; Jacobs et al. 2018; Kobayashi 2006; Lee et al. 2025; Leino et al. 2010; Reinhard and Jacobs 2021a] that denote progress dependencies. Namely, a thread τ is allowed to busy-wait for a thread τ' if all obligations held by τ' are smaller than those held by τ.

Reducing termination to safety, Sassy's approach can, in essence, be carried out in any sufficiently expressive program logic for reasoning about the safety of (higher-order) concurrent programs. Fasse and Jacobs [2024] have chosen to formalize the approach in the Iris program logic, but they did so only on paper *i.e.* they did not leverage the Iris Rocq library. They carried out their case studies against the axiomatization of and tool support for Ghost Signals in VeriFast [Reinhard and Jacobs 2021a,b]. To date, the soundness thereof has not been mechanized.

In this paper, we present Lawyer, which combines the obligations-based reasoning of Sassy [Fasse and Jacobs 2024] with the relational approach of Trillium [Timany et al. 2024]. That is, we show that Trillium's program logic together with an LTS model, which is inspired by Sassy's auxiliary code and state, can be used to reason modularly about the termination of higher-order concurrent programs. This allows us to entirely forego auxiliary code and state and reason directly about a program as opposed to an instrumentation of it. Thus, there is no need for an erasure argument [Fasse and Jacobs 2024], which significantly simplifies the soundness proof and its mechanization.

Besides the small motivating example programs, we also present an abstract specification of fair locks which we instantiate with a concrete implementation — the ticket lock. We also verify a client program against the abstract fair lock specification; by modularity, we obtain a proof of termination of the client linked with the concrete ticket lock implementation. All results presented in this paper, including all examples and case studies, as well as Lawyer's soundness proof, are mechanized in the Rocq Prover [Namakonov et al. 2026a]. For a more detailed comparison of Sassy and Lawyer, including the challenges involved in incorporating Sassy's approach with Trillium's, see §5 and §4.3.

In summary, we make the following contributions:

- The first mechanized higher-order impredicative concurrent separation logic for modular specification and verification of liveness properties of higher-order concurrent programs with shared mutable state.
- A novel obligations model (§2), inspired by Sassy, which abstracts over terminating concurrent programs and which Trillium can be instantiated with. We prove that any fair trace in the obligations model is finite.
- A novel instantiation of the Trillium program logic framework, with new proof rules for manipulating the obligations model, which allow proving refinement to that model (§3)
- Case studies of modular verification of termination, including specification and proof of a fair-lock module and a client thereof (§4). In the main body of the paper, we present a version of the fair lock specification that is reasonably simple to use by clients. For the expert readers, we mention that this specification is derived from a novel logically atomic specification of the fair lock module; our notion of logically atomic specification is inspired by Sassy and extends the logically atomic specifications of Iris to reason about termination. Our logically atomic specification is included in the appendix [Namakonov et al. 2026b].

We discuss further related work in §5 and conclude in §6.

2 Obligations Model

In this section, we define the states and transitions of Obligations model (OM) and establish some important properties of it. Due to the complexity of the problem at hand, *i.e.*, reasoning about termination of higher-order concurrent programs, the states and transitions of OM are also rather complex because they need to capture the intricate interdependence of different program modules of such programs that run in parallel. Following Liang and Feng [2016], we consider two general classes of such programs: *delaying* and *blocking*. To simplify the presentation, we first explain how our model supports delaying, and then extend it to blocking.

Thus, we start in §2.1 by presenting NOM — a *no-obligations* fragment of OM that is still expressive enough to abstract the execution of non-blocking programs that do not fork. Then, in §2.2 we present the full definition of OM that abstracts both non-blocking and blocking programs. To motivate the different parts of OM's state and transitions, in both §2.1 and §2.2 we use a carefully chosen set of (fairly) terminating programs, and for each of them, describe informally how the traces of OM abstract the execution traces of that program. Afterwards, in §2.3 we give the formal definition of the refinement relation between program execution traces and OM traces. Moreover, in [Namakonov et al. 2026b] we describe some additional properties that OM exhibits.

2.1 No-Obligations Model Definition

The formal definition of NOM is given in Figure 1 on Page 5. The model NOM is parameterized by a finite partially ordered set of degrees $(\mathsf{Degree}, \preceq_{\mathsf{D}})$, a set Thread (to be instantiated with the set of thread identifiers of a given language) and a natural number SB; we will describe these later

A state of NOM is a record of type $\left\{\texttt{fuel} : \mathit{FinMultiset}(\text{Barrel}); \texttt{phs} : \text{Thread} \xrightarrow{\text{fin}} \text{Phase}; \texttt{eb} : \mathbb{N}\right\}$,

where $\text{Barrel} \triangleq \text{Phase} \times \text{Degree}$ $\quad \text{Phase} \triangleq \mathit{List}\,\mathbb{N}$ $\quad \pi_1 \preceq_{\text{P}} \pi_2 \triangleq \exists \pi'.\, \pi_2 = \pi' +\!\!+ \pi_1$

Small transitions of NOM

$$\textsc{noms-exchanges-fuel}\ \frac{(\pi, d) \in \delta.\texttt{fuel} \qquad d' \prec_{\text{D}} d \qquad n \le \delta.\texttt{eb}}{\delta \underset{\text{NOM}}{\dashrightarrow} \delta[\texttt{fuel} := \delta.\texttt{fuel} \setminus \{1 \cdot (\pi, d)\} \uplus \{n \cdot (\pi, d')\}]} \qquad \textsc{noms-increases-eb}\ \frac{}{\delta \underset{\text{NOM}}{\dashrightarrow} \delta[\texttt{eb} := \delta.\texttt{eb} + 1]}$$

Fuel burning and the full transition of NOM

$$\textsc{nom-burns-fuel}\ \frac{\delta.\texttt{phs}[\tau] = \pi_\tau \qquad \pi \preceq_{\text{P}} \pi_\tau \qquad (\pi, d) \in \delta.\texttt{fuel}}{\delta \overset{\tau}{\underset{\text{NOM}}{\rightsquigarrow}} \delta[\texttt{fuel} := \delta.\texttt{fuel} \setminus \{1 \cdot (\pi, d)\}]} \qquad \delta_1 \underset{\text{NOM}}{\overset{\tau}{\to}} \delta_2 \triangleq \exists n \le \text{SB}, \delta'.\ \delta_1 \underset{\text{NOM}}{\dashrightarrow}^{n} \delta' \wedge \delta' \overset{\tau}{\underset{\text{NOM}}{\rightsquigarrow}} \delta_2$$

Fig. 1. No-Obligations model (parameterized by Degree, Thread and SB)

on. The model states $\delta : \text{NOM}_{\text{St}}$ and transitions $\delta_1 \xrightarrow[\text{NOM}]{\tau:\text{Thread}} \delta_2$ are chosen so that the traces of NOM abstract the execution traces of terminating non-blocking programs. This explains the choice of labels: a transition labeled with τ corresponds to an execution step of thread τ.

As working with OM involves operations on multisets, we introduce a number of notations here. We write $\{n \cdot x\}$ to denote a multiset with a single element x whose multiplicity is n. We write $M_1 \uplus M_2$ for multiset union and $M_1 \setminus M_2$ for multiset difference, which respectively increase the multiplicity of added elements, and decrease the multiplicity of the removed elements—multiplicities cannot become negative, of course. Also, if M is a multiset, we use a shorthand $x \in M$ whenever $\{1 \cdot x\} \subseteq M$, where the subset relation on multisets, written $M_1 \subseteq M_2$, means that the multiplicity of every element in M_1 is less than or equal to that in M_2.

Independent Termination with Statically-Known Bound. We begin with the example in Figure 2. There and in the examples below, ‖ denotes the parallel composition of two threads already present in the program's starting configuration; later, we show how our approach extends to unstructured, fork-based concurrency. Assuming that the reference l initially stores a non-negative integer N, both threads terminate in a constant amount of time: the left thread terminates in exactly 1 step, whereas the right one takes up to $K_{\texttt{decr}} * N$ steps to terminate — the constant $K_{\texttt{decr}}$ over-approximates the number of steps of a single `decr` iteration (we take 10 in our Rocq mechanization). We can abstract an execution of this example by transitions of NOM with $\text{Degree} \triangleq \{d_0\}$[1] and $\text{SB} \triangleq 0$. For simplicity, until §2.2 we also assume that $\text{Thread} \triangleq \mathbb{N}$.

```
let rec decr l =
  let c = !l in
  if c == 0 then ()
  else l := c - 1; decr l
in !l ‖ decr l
```

Fig. 2. Statically-known Bound

For simple examples such as Figure 2, a NOM state represents an assignment of *fuel* to threads. In that case, every element (a *barrel*) of `fuel` multiset is marked as belonging to one of the threads via `phs` mapping (we elaborate on this mapping below). The `eb` state component is irrelevant for this example, and we set it to 0. Therefore, we consider NOM states of the form $\{\texttt{fuel} := \biguplus_{\tau \in T} \{N_\tau \cdot ([\tau], d_0)\}; \texttt{phs} := \lambda\tau \Rightarrow [\tau]; \texttt{eb} := 0\}$ for some choices of N_τ, where T is the set of thread identifiers used in given program (here, they are $\tau_l \triangleq 0$ and $\tau_r \triangleq 1$ denoting the left and right thread). Omitting the (almost) identity `phs` and irrelevant `eb`, an NOM state abstracting the

[1]We denote the single element of that set as d_0 to be uniform with other examples below.

state of the example program in Figure 2 can thus be written for some n_l, n_r as

$$\{\texttt{fuel} := \{n_l \cdot ([\tau_l], d_0)\} \uplus \{n_r \cdot ([\tau_r], d_0)\}\}$$

This means, in turn, that a trace of the program's execution can be abstracted by a trace of NOM by taking n_l and n_r to initially be (an overapproximation of) the number of steps that the left and right thread take, respectively. In its simplest form, taking $n = 0$, $\xrightarrow[\text{NOM}]{\tau}$ reduces to $\overset{\tau}{\rightsquigarrow}_{\text{NOM}}$ (see Figure 1). Hence, according to NOM-BURNS-FUEL, a $\overset{\tau}{\rightsquigarrow}_{\text{NOM}}$ transition consumes one barrel assigned to τ. With $\rightsquigarrow_{\text{NOM}}$ transitions only, a thread's fuel decreases every time it takes a step; therefore, the traces of the simplified model always terminate in $n_l + n_r$ steps. (We drop the thread's name from arrows like $\rightsquigarrow_{\text{NOM}}$ and $\xrightarrow[\text{NOM}]{}$ when it is clear from the context or irrelevant.)

As we shall see later, a precise correspondence between the program execution and a model trace is established by conducting a proof within the program logic that we present in §3. At every step of that proof, we intuitively need to show that the given thread has enough fuel to continue its execution. For that to hold, we choose the initial state of the obligations model to have $n_l \triangleq 1$ and $n_r \triangleq K_{\texttt{decr}} * N$ and, moreover, we keep track of the fuel at every step. Upon completion of the proof, it will be the case that the program refines a trace of the chosen model (see §2.3), which in turn implies that the program execution terminates. In fact, our Rocq formalization proves that the upper bound on the execution time is linear wrt. N.

Termination with Runtime-Determined Bound. Consider now the example in Figure 3, where `nondet` is a function which returns an arbitrary natural number, and `decr` is as in Figure 2. In this example, the execution time depends on the return value of `nondet` (its implementation forks a thread internally and requires increasing `eb` — it is irrelevant for this example and will be discussed later). Thus, we cannot directly repeat the process used for Figure 2 as the number of barrels to provide initially cannot be determined statically.

```
let k = nondet () in
let l = ref k in
decr l
```

Fig. 3. Dynamic Bound

The key idea for verifying Figure 3 is to postpone providing any concrete amount of fuel for `decr` until `nondet` is evaluated. To justify it, one can initially provide a single "larger" barrel which can be exchanged for a number of "smaller" barrels once that number is known. More formally, we stratify fuel barrels by their degree. Let Degree $\triangleq \{d_0, d_1, d_2\}$ such that $d_0 \prec_{\text{D}} d_1 \prec_{\text{D}} d_2$, and let SB $\triangleq 1$; moreover, let $\tau \triangleq 0$ denote the only thread present in the program. Importantly, if the relation Degree on $\prec_{\text{D}}$ is well-founded, the multiset $\{n \cdot d_0\}$ is smaller than $\{d_1\}$ for any n [Dershowitz and Manna 1979].

We defer verifying `nondet` until §2.2 and assume that after `nondet` evaluates to some k, the program state corresponds to the NOM state $\{\texttt{fuel} := \{1 \cdot ([\tau], d_2)\}; \texttt{eb} := k'\}$ for some $k' > k$ (we explain the `eb` component later). At this point, knowing that we pass to `decr` exactly the number k (stored in memory location l), we will exchange the one barrel of $([\tau], d_2)$ fuel with $k + 1$ barrels of $([\tau], d_1)$ using the NOMS-EXCHANGES-FUEL transition: k for passing to `decr` and 1 for the reference creation step. For this purpose, we take advantage of the fact that we can take up to SB many *small transitions* $\dashrightarrow_{\text{NOM}}$. After that, inside `decr`, on each of the k iterations we similarly exchange a barrel of $([\tau], d_1)$ for $K_{\texttt{decr}}$ barrels of $([\tau], d_0)$ and burn them during that iteration. (Note that the earlier proof explained above for `decr` was a simplified version of the proof that does not take advantage of exchanging fuel as we explained here.)

Delaying Other Threads and Fuel Sharing. The example in Figure 4 showcases delaying. This example features two threads, both attempting to increment the shared counter `c` using a compare-and-set (`CAS`) operation. Depending on the scheduler, it can happen that one thread's `CAS` operation

succeeds after the other thread has read `c`, but before it has attempted to update it via its `CAS` operation. The other thread's `CAS` thus fails, leading to a recursive call (delay).

Looking at the example in Figure 4 as a whole, we can see that, in the worst case, one thread is delayed by the other—and that at most once. We can generalize this observation: if we were to run n instances of `incr` concurrently (instead of the two in Figure 4), each thread's `CAS` operation fails at most $n - 1$ times. This poses a challenge for modular verification. When reasoning about `incr` in isolation (as we do in the Rocq development), we must show its termination regardless of the amount of concurrent instances of `incr`.

```
let c = ref 0 in
let rec incr () =
  let v = !c in
  if CAS(c, v, v+1)
  then ()
  else incr ()
in
  incr () ‖ incr ()
```

Fig. 4. Delaying example

Intuitively, what we need for reasoning about termination of `incr` is to *share* fuel between the two threads. The idea is that we initially assume that we have enough fuel to run all instances of `incr`, that is to say, proportional to the maximum number `CAS` operations run (both successful and unsuccessful ones). Once the `CAS` operation inside an `incr` succeeds, *the rest* of the fuel of that instance is shared with the other instances of `incr` for which our success in performing the `CAS` operation may necessitate an extra recursive call [Jacobs et al. 2018]. Technically, in the program logic, this sharing is mediated by an invariant that keeps track of the value of the current counter and how many threads have read this value prior to performing their `CAS` operation. Among these, the first that reaches its `CAS` operation will have it succeed and will be required to share the rest of its fuel (relinquish it to the invariant) with the other threads. All the other threads in the system that have read the same counter value will then, upon having their `CAS` operation failing, know that another thread's `CAS` has succeeded, and thus know there is more fuel put aside for them by the thread whose `CAS` succeeded. This fuel sharing mechanism is the key for verifying delaying [Liang and Feng 2016] programs.

The scheme explained above for exchanging fuel would only work if one thread's fuel could be used by other threads. However, while verifying Figure 7 in §2.2, we will explain why doing it naively would not be sound when `NOM` is extended to support blocking programs. Intuitively, verifying blocking programs requires a mechanism for *generating more fuel*, and with unrestricted fuel sharing it might lead to our model allowing self-fueling loops. We therefore need a way to support restricted and sound fuel sharing.

Our solution uses the mechanism of phases [Reinhard and Jacobs 2021a,b] which allow sharing fuel *among threads with a shared ancestor*; in §2.2, we will see how this approach ensures soundness of sharing generated fuel. Phases relax the relationship between threads and their fuel: instead of assigning fuel to threads, we assign phases to threads (with the `phs` component of `NOM` state) and fuel to phases. A phase is a list of 0's and 1's, intuitively encoding a position in the threads' fork tree. We will consider forking in more detail in §2.2; in short, the main thread of the program initially has phase *nil*, and when a thread with phase π_τ forks, its phase becomes $0 :: \pi_\tau$, and the new forked thread is assigned phase $1 :: \pi_\tau$. As per NOM-BURNS-FUEL, when a thread τ with a phase π_τ takes a step, we require fuel of some phase π to be burned such that $\pi \preceq_{\mathsf{P}} \pi_\tau$ (the *phase order* $\preceq_{\mathsf{P}}$ is defined in Figure 1). Thus, if τ is provided with some fuel of phase π, it can either consume it itself or share it with any other thread τ' with phase $\pi_{\tau'}$ such that $\pi \preceq_{\mathsf{P}} \pi_{\tau'}$.

For Figure 4, in our Rocq development we choose Degree $\triangleq \{d_0, d_1\}$ and show that a single call to `incr` can be verified separately by providing it with $2 \cdot (\pi, d_1)$. Here, π must be a phase compatible (in the phase order) with phases of all threads that call `incr`. Since we do not consider forking yet, we assume that the two threads τ_l and τ_r are initially assigned the phases $[0]$ and $[1]$ correspondingly. In that case, we can choose $\pi \triangleq \mathit{nil}$ and provide $\{(2 * 2) \cdot (\mathit{nil}, d_1)\}$ fuel to verify Figure 4.

Note that neither thread in Figure 4 will block if another one is never scheduled. Indeed, our Rocq formalization shows that this program terminates under any, possibly unfair scheduler.

Tracking the Exchange Bound. To complete the explanation of NOM transitions, we here explain the remaining NOMS-INCREASES-EB. Note that the transition NOMS-EXCHANGES-FUEL bounds the amount of exchanged fuel by the current value of eb — the technical reason for requiring a bound is explained in §6. To make our approach to proving termination more flexible, we do not enforce a globally fixed bound for the entire program. Rather, we allow eb to be increased dynamically through the transition NOMS-INCREASES-EB. Indeed, doing so was needed to verify Figure 3 where we assumed that $\mathtt{eb} > k$ holds after `nondet` evaluates to some k. We verify `nondet` in §2.2; right now, the most important part is that it is implemented by repeatedly increasing a counter (see `incr_loop` in Figure 6). Thus, while verifying `nondet`, we use NOMS-INCREASES-EB on every iteration of its increment loop, which ensures that when `nondet` produces some result k, the current value of eb is at least k. One extra NOMS-INCREASES-EB ensures that $\mathtt{eb} > k$ which is needed to verify Figure 3.

2.2 Obligations Model Definition

In this section, we define the full obligations model OM by extending NOM with support for blocking concurrency. The formal definition of OM is given in Figure 5 on Page 9. Besides Degree, Thread and SB, the model parameterized by a finite partially ordered set of levels $(\mathsf{Level}, \preceq_{\mathsf{L}})$. The transition $\xrightarrow[\mathtt{OM}]{}$ of OM is defined similarly to $\xrightarrow[\mathtt{NOM}]{}$, additionally supporting a forking transition OM-FORKS. In the examples below, we illustrate the new components of the OM state and new small transitions.

Busy-Waiting on Other Threads. Consider the example in Figure 6. The program consists of two threads and assumes the existence of two pre-allocated locations `f`, which is initially **`false`**, and `n`, which is initially 0. The right thread τ_r busy-waits for the left thread τ_l to set `f` to **`true`**. If the left thread is never scheduled, the right thread will loop forever because it is blocked on `f` being set. With fuel alone, proving termination of this busy-waiting pattern is impossible: we cannot pick a decreasing measure justifying termination of the right thread as a function of the current values of `f` and `n`. In fact, the number of steps this program takes depends on the schedule. Thus, we need to exploit the scheduler's fairness to show that eventually the left thread performs the write to `f`, which in turn implies that the right thread terminates.

```
let rec incr_loop f n =
  if (!f == true) then ()
  else FAA(n, 1); incr_loop f n
in
f := true; !n ‖ incr_loop f n
```

Fig. 6. Blocking example

One of the purposes of our obligations model OM, and one of the reasons for its complexity, is avoiding such explicit reasoning about the scheduler. That is, OM provides an abstraction, namely obligations, for actions eventually performed by one thread and waited for by others. To use obligations for proving termination of the program in Figure 6, we instantiate OM with $\mathsf{SB} \triangleq 2$, $\mathsf{Level} \triangleq \{\ell_0\}$, $\mathsf{Degree} \triangleq \{d_0\}$. In our example, with a single busy-waiting loop, and hence a single obligation level, ℓ_0, the partial order on Level is trivial, but more complicated examples can be supported by choosing an appropriate well-founded $\preceq_{\mathsf{L}}$ on obligation levels. To simplify the explanation, we will assume that some components of the initial state of the model are initialized to specific values. Later on, we will explain how their values can be set dynamically instead. Specifically, we assume that the initial state of the Obligations model is

$$\left\{\begin{array}{l}\mathtt{fuel} := \{2 \cdot ([0], d_0)\} \uplus \{1 \cdot ([1], d_0)\}; \mathtt{sigs} := \{s_f \mapsto (\ell_0, \mathsf{False})\}; \mathtt{eb} := 0; \\ \mathtt{obls} := \{\tau_l \mapsto \{s_f\}; \tau_r \mapsto \emptyset\}; \mathtt{eps} := \{(s_f, [1], d_0)\}; \mathtt{phs} := \{\tau_l \mapsto [0]; \tau_r \mapsto [1]; \}\end{array}\right\} \quad \text{(blocking-initial)}$$

A state of OM is a well-formed (see [Namakonov et al. 2026b]) record of type {fuel; sigs; obls; eps; phs; eb} with fields' types as follows:

$$\begin{array}{ll}
\texttt{fuel} : \mathit{FinMultiset}(\mathsf{Barrel}) & \mathsf{Barrel} \triangleq \mathsf{Phase} \times \mathsf{Degree} \\
\texttt{sigs} : \mathsf{SignalId} \xrightarrow{\text{fin}} \mathsf{SignalState} & \mathsf{ExpectPerm} \triangleq \mathsf{SignalId} \times \mathsf{Phase} \times \mathsf{Degree} \\
\texttt{obls} : \mathsf{Thread} \xrightarrow{\text{fin}} \mathit{FinSet}(\mathsf{SignalId}) & \mathsf{SignalId} \triangleq \mathbb{N} \\
\texttt{eps} : \mathit{FinSet}(\mathsf{ExpectPerm}) & \mathsf{SignalState} \triangleq \mathsf{Level} \times \mathbf{B} \\
\texttt{phs} : \mathsf{Thread} \xrightarrow{\text{fin}} \mathsf{Phase} & \mathsf{Phase} \triangleq \mathit{List}\,\mathbb{N} \\
\texttt{eb} : \mathbb{N} & \pi_1 \preceq_{\mathrm{P}} \pi_2 \triangleq \exists \pi'.\, \pi_2 = \pi' +\!\!+ \pi_1
\end{array}$$

Small transitions of OM

OMS-EXCHANGES-FUEL

$$\frac{(\pi, d) \in \delta.\texttt{fuel} \qquad d' \prec_{\mathrm{D}} d \qquad n \le \delta.\texttt{eb}}{\delta \xrightarrow[\texttt{OM}]{} \delta[\texttt{fuel} := \delta.\texttt{fuel} \setminus \{1 \cdot (\pi, d)\} \uplus \{n \cdot (\pi, d')\}]}$$

OMS-INCREASES-EB

$$\frac{}{\delta \xrightarrow[\texttt{OM}]{} \delta[\texttt{eb} := \delta.\texttt{eb} + 1]}$$

OMS-CREATES-SIGNAL

$$\frac{\tau \in \operatorname{dom} \delta.\texttt{obls} \qquad s \triangleq \max(\operatorname{dom} \delta.\texttt{sigs}) + 1 \qquad l \in \mathsf{Level}}{\delta \xrightarrow[\texttt{OM}]{} \delta[\texttt{sigs} := \delta.\texttt{sigs}[s := (l, \mathsf{False})]][\texttt{obls} := \delta.\texttt{obls}[\tau := \delta.\texttt{obls}[\tau] \cup \{s\}]]}$$

OMS-SETS-SIGNAL

$$\frac{s \in \delta.\texttt{obls}[\tau] \qquad \delta.\texttt{sigs}[s] = (l, \_) \qquad \tau \in \operatorname{dom} \delta.\texttt{obls}}{\delta \xrightarrow[\texttt{OM}]{} \delta[\texttt{sigs} := \delta.\texttt{sigs}[s := (l, \mathsf{True})]][\texttt{obls} := \delta.\texttt{obls}[\tau := \delta.\texttt{obls}[\tau] \setminus \{s\}]]}$$

OMS-CREATES-EP

$$\frac{\delta.\texttt{phs}[\tau] = \pi_\tau \qquad \pi \preceq_{\mathrm{P}} \pi_\tau \qquad (\pi, d) \in \delta.\texttt{fuel} \qquad s \in \operatorname{dom} \delta.\texttt{sigs} \qquad d' \prec_{\mathrm{D}} d}{\delta \xrightarrow[\texttt{OM}]{} \delta[\texttt{fuel} := \delta.\texttt{fuel} \setminus \{1 \cdot (\pi, d)\}][\texttt{eps} := \delta.\texttt{eps} \cup \{(s, \pi, d')\}]}$$

OMS-EXPECTS-EP

$$\frac{\delta.\texttt{phs}[\tau] = \pi_\tau \qquad \pi \preceq_{\mathrm{P}} \pi_\tau \qquad \delta.\texttt{sigs}[s] = (l, \mathsf{False}) \qquad (s, \pi, d) \in \delta.\texttt{eps} \qquad \forall o \in \delta.\,\texttt{obls}[\tau].\, l \prec_{\mathrm{L}} \delta.\texttt{sigs}[o].1}{\delta \xrightarrow[\texttt{OM}]{} \delta[\texttt{fuel} := \delta.\texttt{fuel} \uplus \{1 \cdot (\pi_\tau, d)\}]}$$

Fuel burning, fork and the full transition of OM

OM-BURNS-FUEL

$$\frac{\delta.\texttt{phs}[\tau] = \pi_\tau \qquad \pi \preceq_{\mathrm{P}} \pi_\tau \qquad (\pi, d) \in \delta.\texttt{fuel}}{\delta \overset{\tau}{\underset{\texttt{OM}}{\rightsquigarrow}} \delta[\texttt{fuel} := \delta.\texttt{fuel} \setminus \{1 \cdot (\pi, d)\}]}$$

OM-FORKS

$$\frac{\begin{array}{c}\delta.\texttt{phs}[\tau] = \pi_\tau \qquad \tau' \notin \operatorname{dom} \delta.\texttt{phs} \\ \delta.\texttt{obls}[\tau] = U \cup V \qquad U \# V \\ P' \triangleq \delta.\texttt{phs}[\tau := 0 :: \pi_\tau][\tau' := 1 :: \pi_\tau] \\ O' \triangleq \delta.\texttt{obls}[\tau := U][\tau' := V]\end{array}}{\delta \xrightarrow[\texttt{OM}]{\tau,\tau'} \delta[\texttt{obls} := O'][\texttt{phs} := P']}$$

$$\delta_1 \xrightarrow[\texttt{OM}]{\tau} \delta_2 \triangleq \exists n, \delta', \delta''.\, n \le \mathsf{SB} \wedge \delta_1 \xrightarrow[\texttt{OM}]{}^{n} \delta' \wedge \delta' \overset{\tau}{\underset{\texttt{OM}}{\rightsquigarrow}} \delta'' \wedge (\delta'' = \delta_2 \vee \exists \tau'.\, \delta'' \xrightarrow[\texttt{OM}]{\tau,\tau'} \delta_2)$$

Fig. 5. Obligations model (parameterized by Level, Degree, Thread and SB)

where $s_f \triangleq 0 : \mathsf{SignalId}$ is a *signal* and the obligation of thread τ_l being tied to that signal.

An unset signal establishes the existence of a thread with an obligation to set that signal. The `sigs` field of a state in `OM` associates every signal with its current *signal state*. An obligation can only be fulfilled by performing an OMS-SETS-SIGNAL small transition. Taking this transition removes the thread obligation and updates the signal's state. Thus, by tying a physical state change to the setting of a signal (with an invariant), we express that the corresponding action must eventually happen. Correspondingly, a trace of `OM` terminates if all threads fulfill their obligations.

In our example, s_f has level ℓ_0 (the only choice for this example), and initially the action it represents has not been performed yet (indicated by `False` in the signal state). Thread τ_l has the obligation to set s_f and, enforced by an invariant, must therefore set `f` eventually. If it did not, thread τ_l would need to diverge or terminate with obligations left—both ruled out by refining `OM`.

Intuitively, burning fuel serves as the termination measure, *i.e.* all steps of `OM` burn some fuel. Now, whereas the termination of the left thread in Figure 6 is obvious, the termination of the right thread, when considered in isolation, is not. Indeed, it relies on the fact that the left thread will eventually set the flag location `f`. This aspect of the behavior of the right thread is captured abstractly at the level of `OM` with the signal s_f playing the role of the location `f`. Still, the number of steps the right thread takes depends on the scheduler: it must busy-wait until `f` is set. Therefore, there is no finite bound on the number of steps (and fuel) the right thread takes, even when we only consider fair schedulers. Thus, following Reinhard and Jacobs [2021a,b], `OM` allows fuel to be replenished as long as we wait for (are blocked on) a signal to be set. In `OM`, replenishing the fuel is supported by the small transition OMS-EXPECTS-EP (*expect*ing the signal to be set eventually).

In the example in Figure 6, we take two OMS-EXPECTS-EP transitions every time the right thread, τ_r, reads **`false`** from `f`. This generates two barrels of $([1], d_0)$ fuel, which in turn are spent for model transitions corresponding to the `FAA` and the read of the location `f` in the next recursive call. To enable these OMS-EXPECTS-EP transitions, the initial state (blocking-initial) includes the initial *expectation permission* $(s_f, [1], d_0)$. Restricting when it is possible to generate fuel is necessary so that any fair `OM` trace is finite (cf. Reinhard and Jacobs [2021a,b]). The side-condition on levels, *i.e.*, $\forall o \in \mathtt{obls}[\tau].\ l \prec_{\mathsf{L}} \delta.\mathtt{sigs}[o].1$, in OMS-EXPECTS-EP rules out potential circular waiting among obligations which is crucial for guaranteeing termination — note that in Lemma 2.3 we require $\preceq_{\mathsf{L}}$ to be well-founded. In case of Figure 6, this side-condition holds trivially since τ_r has no obligations.

Forking Threads. We now consider the implementation of `nondet` in Figure 7. Here, the main thread starts the increment loop from above in a new thread using fork and then continues execution. In `OM`, the step of the main thread τ that forks the child thread τ' is matched by a forking transition, $\xrightarrow[\mathtt{OM}]{\tau,\tau'}$. As can be seen in $\xrightarrow[\mathtt{OM}]{\tau}$, this kind of transition can be taken after burning fuel. Forking transitions update the current phase of τ and assign the forked thread τ' a sibling phase; two phases are called sibling phases if they are of the form $0 :: \pi$ and $1 :: \pi$. Moreover, we allow transferring some of the forking thread's obligations to the forkee.

```
let nondet () =
  let f = ref false in
  let cnt = ref 0 in
  (* incr_loop as in Figure 6 *)
  fork (incr_loop f cnt);
  f := true; !cnt
```

Fig. 7. Fork example

In the examples we have considered above, we used a static number of threads, which simplified the description of the model states and transitions, because the initial state we considered for `OM` could simply refer to thread identifiers. This simpler treatment also allowed us to assume that the necessary signals and obligations exist in the initial state of `OM` we considered. To support dynamic creation of threads, we also need to support dynamic creation of signals, which we support through the transition OMS-CREATES-SIGNAL, and dynamic creation of expectation permissions, which we

support through the transition OMS-CREATES-EP. The transition OMS-CREATES-SIGNAL creates a fresh unset signal at the given level and assigns it as an obligation to the given thread. The transition OMS-CREATES-EP consumes a barrel of some degree d to create an expectation permission for an existing signal, with some lower degree d', which in turn allows generating fuel at degree d' using the transition OMS-EXPECTS-EP. After the forking and creation of the signal and expect permission, verification of Figure 7 follows closely that of Figure 6.

Note that without the phases (*i.e.* allowing unrestricted fuel sharing) we could (unsoundly) verify a variation of Figure 7 where the main thread also runs `incr_loop` instead of setting the flag. For that, the forked thread would produce fuel by waiting for the flag to be set and then share it with the main thread, justifying its looping. To prevent that, OMS-EXPECTS-EP generates fuel of the thread's *current* phase, which is always phase order-incompatible with other threads' phases. With that, generated fuel can only be burned by the thread itself and the threads it will fork in the future.

2.3 Refinement Relation

In this section, we define the operational semantics of $\lambda_{\text{ref}}^{\text{conc}}$, the concurrent higher-order imperative programming language we work with in this paper. Afterwards, we formally define the refinement relation between program execution traces and OM traces, and we show how such a refinement allows proving fair termination of programs. In the following section, we present the program logic of Lawyer, which we use to establish such refinements.

In the rest of the section, we use the following notations. Given a trace tr of some LTS, $tr[i]$ and $tr\langle i\rangle$ denote its ith state and label, respectively. In general, the traces we work with may be finite or infinite, but always non-empty. If tr is finite, $tr.\text{last}$ denotes its last state. Then, $tr :\langle \ell\rangle: s$ denotes the trace tr extended with the transition from $tr.\text{last}$ to the state s with the label ℓ — this notation, of course, only makes sense, and is used when tr is a finite trace. We write $\text{dom}(tr)$ for the set of indices i for which $tr[i]$ is defined, *i.e.*, integers between 0 and the length of tr. Finally, we say that a trace of LTS $\mathcal{M}$ is *valid* if it denotes a sequence of $\mathcal{M}$ transitions:

$$\mathit{valid}_{\mathcal{M}}(tr) \triangleq \forall i.\, i+1 \in \text{dom}\, tr \implies tr[i] \xrightarrow[\mathcal{M}]{tr\langle i\rangle} tr[i+1]$$

Operational Semantics of $\lambda_{\text{ref}}^{\text{conc}}$. The language $\lambda_{\text{ref}}^{\text{conc}}$ is an ML-like programming language with higher-order references and concurrency. An excerpt of its operational semantics is shown in Figure 8; see our Rocq development for further details. The operational semantics is defined as a binary relation over configurations, which are pairs of a thread pool and a heap. The relation non-deterministically chooses a thread and "evaluates" the program in that thread by decomposing it into an evaluation context E and an expression e which is then evaluated as per the top-level per-thread reduction relation. Note how the evaluation contexts determine a right-to-left call by value reduction strategy. We also remark that the operational semantics, *i.e.*, the configuration reduction relation, forms an LTS as reductions are labeled by the thread being reduced.

Refinement. To define the notion of refinement between execution traces of $\lambda_{\text{ref}}^{\text{conc}}$ and traces of OM, we first define a relation $\simeq$ between the configurations of $\lambda_{\text{ref}}^{\text{conc}}$ and states of OM. This relation requires that the threads represented in the OM state are exactly those in the configuration's thread pool. Moreover, it requires that the threads carrying obligations have not terminated yet. Formally, this relation is defined as follows:

Definition 1 (The relation between configurations and model states). *The relation between a configuration c and an* OM *state δ, written $c \simeq \delta$, is defined as follows:*

$$c \simeq \delta \triangleq \mathit{sameThreads}(c, \delta) \wedge \mathit{liveThreads}(c, \delta)$$

Syntax

$$
\begin{array}{llll}
x, y, f \in \mathit{Var} & & l \in \mathit{Loc} & n \in \mathbb{Z} \\
& \circledcirc & ::= & + \mid - \mid * \mid = \mid < \mid \cdots \\
\mathit{Val} & v & ::= & () \mid \texttt{true} \mid \texttt{false} \mid n \mid l \mid (v, v) \mid \texttt{rec}\ f\ (x) := e \mid \ldots \\
\mathit{Expr} & e & ::= & x \mid v \mid e \circledcirc e \mid \texttt{ref}\ (e) \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{fork}\ (e) \mid\ !\, e \mid e\, e \mid \ldots \\
\mathit{ECtx} & E & ::= & \bullet \mid e \circledcirc E \mid E \circledcirc v \mid \texttt{if}\ E\ \texttt{then}\ e\ \texttt{else}\ e \mid (e, E) \mid (E, v) \mid e\, E \mid E\, v \mid \ldots \\
\mathit{Heap} & h & \in & \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Val} \\
\mathit{ThreadPool} & \mathcal{E} & \in & \mathrm{Thread} \xrightarrow{\text{fin}} \mathit{Expr} \\
\mathit{Config} & c & ::= & (\mathcal{E}, h)
\end{array}
$$

Configuration reduction (with $\rightsquigarrow \subseteq (\boldsymbol{Expr} \times \boldsymbol{Heap})^2$ omitted; see [Namakonov et al. 2026b])

$$
\frac{(e, h) \rightsquigarrow (e', h')}{(\mathcal{E}[\tau \mapsto E[e]], h) \xrightarrow{\tau} (\mathcal{E}[\tau \mapsto E[e']], h')}
\qquad
\frac{\tau' \notin \mathrm{dom}(\mathcal{E}) \cup \{\tau\} \qquad \mathcal{E}' \triangleq \mathcal{E}[\tau \mapsto E[()]][\tau' \mapsto e]}{(\mathcal{E}[\tau \mapsto E[\texttt{fork}\,(e)]], h) \xrightarrow{\tau} (\mathcal{E}', h)}
$$

Fig. 8. An excerpt of operational semantics of the language $\lambda^{\text{conc}}_{\text{ref}}$

where

$$
\mathit{sameThreads}(c, \delta) \triangleq \mathrm{dom}\, c.1 = \mathrm{dom}\, \delta.\texttt{obls} = \mathrm{dom}\, \delta.\texttt{phs} \qquad \mathit{threadActive}(\tau, c) \triangleq c.1[\tau] \notin \mathit{Val}
$$

$$
\mathit{liveThreads}(c, \delta) \triangleq \forall \tau.\ \mathit{hasObls}(\tau, \delta) \implies \mathit{threadActive}(\tau, c) \qquad \mathit{hasObls}(\tau, \delta) \triangleq \delta.\texttt{obls}[\tau] \neq \emptyset
$$

Now, we define the refinement relation between program execution traces and OM traces based on the relation between program configurations and states of OM in Definition 1:

Definition 2 (The refinement relation on finite traces). *Let etr be a finite execution trace of $\lambda^{\text{conc}}_{\text{ref}}$ and let mtr be a finite* OM *trace. The refinement relation between etr and mtr, written $\xi^{\lambda}_{\texttt{OM}}(\mathit{etr}, \mathit{mtr})$ is defined as follows:*

$$
\begin{aligned}
\xi^{\lambda}_{\texttt{OM}}(\mathit{etr}, \mathit{mtr}) \triangleq\ & \mathit{valid}_{\lambda^{\text{conc}}_{\text{ref}}}(\mathit{etr}) \wedge \mathit{valid}_{\texttt{OM}}(\mathit{mtr}) \wedge \mathrm{dom}\, \mathit{etr} = \mathrm{dom}\, \mathit{mtr} \wedge \\
& (\forall i \in \mathrm{dom}\, \mathit{etr}.\ \mathit{etr}[i] \simeq \mathit{mtr}[i]) \wedge (\forall i.\ i + 1 \in \mathrm{dom}\, \mathit{etr} \implies \mathit{etr}\langle i \rangle = \mathit{mtr}\langle i \rangle)
\end{aligned}
$$

In order to instantiate Trillium with the refinement relation $\xi^{\lambda}_{\texttt{OM}}$, we need to show it is relatively image-finite [Timany et al. 2024].

Lemma 2.1. *$\xi^{\lambda}_{\texttt{OM}}$ is relative image-finite:* $\forall \mathit{etr}, \tau_1, c, \mathit{mtr}.\ \mathrm{finite}\{(\tau_2, \delta) \mid \xi^{\lambda}_{\texttt{OM}}(\mathit{etr} :\langle \tau_1 \rangle: c, \mathit{mtr} :\langle \tau_2 \rangle: \delta)\}$

The Adequacy Theorems of Trillium and Lawyer. By adequacy theorem, here we mean the theorem that states and establishes the soundness of the program logic. Here we explain these adequacy theorems informally, relegating the precise statements to our Rocq development. Intuitively, **the adequacy theorem of Trillium** states that proving a Hoare triple in the program logic of Trillium establishes a refinement between the program and the LTS model that Trillium is instantiated with. This refinement relation states that for any possibly infinite execution of the program, there is a corresponding possibly infinite trace of the LTS model related to it by the refinement relation in Definition 3. The program logic of Lawyer is built on top of the instantiation of Trillium with the OM model (*it is not merely an instantiation of Trillium*). **The adequacy theorem of Lawyer** (formally stated in the supplementary material [Namakonov et al. 2026b]) states that proving a Hoare triple in the program logic of Lawyer establishes that all fairly scheduled executions of the program terminate. This theorem follows from the adequacy theorem of Trillium as we will explain below.

Definition 3 (Refinement of possibly infinite traces). *Let etr be a (possibly infinite) execution trace and let mtr be a (possibly infinite) obligations model trace of the same length.*[2] *Then, the refinement relation on possibly infinite traces, written* $etr \hat{\preccurlyeq}_{\xi^{\lambda}_{\mathtt{OM}}} mtr$, *holds if all corresponding finite prefixes of etr and mtr are related by the refinement relation* $\xi^{\lambda}_{\mathtt{OM}}$.

In the rest of this section, we will show that whenever a trace *etr* of a program refines a trace *mtr* of OM, and furthermore *etr* is fairly scheduled, then *etr* is finite, *i.e.*, it terminates. To do this, we first define what fairness means for an arbitrary LTS (Definition 4) and show that whenever *etr* refines *mtr* and *etr* is fair, so is *mtr* (Lemma 2.2). Then, we show that any fair trace *mtr* of OM is necessarily finite (Lemma 2.3). Since *etr* refines *mtr*, by Definition 3, *etr* and *mtr* must have the same length, thus, *etr* must also be finite.

Definition 4 (Fairness). *Given an LTS* $\mathcal{M}$ *and an enabledness predicate* $\mathtt{E} : \mathtt{Label}(\mathcal{M}) \to \mathtt{State}(\mathcal{M}) \to$ *Prop, a trace tr of* $\mathcal{M}$ *is fair with respect to* E, *written* $FairTrace_{\mathtt{E}}(tr)$, *iff:*

$$FairTrace_{\mathtt{E}}(tr) \triangleq \forall \ell, i.\, \mathtt{E}(\ell, tr[i]) \implies \exists j \geq i.\, (\neg \mathtt{E}(\ell, tr[j]) \vee tr\langle j \rangle = \ell)$$

Based on this we define fairness for program and OM *traces as follows (see Definition 1 for definitions of threadActive and hasObls):*

$$fairExec(etr) \triangleq FairTrace_{threadActive}(etr) \qquad oblsFair(mtr) \triangleq FairTrace_{hasObls}(mtr)$$

The definition of $FairTrace_{\mathtt{E}}$ above is equivalent to weak fairness for infinite traces, *i.e.*, if a transition by a given label is continuously enabled, it is eventually taken. For finite traces, on the other hand, it implies that in the final state of the trace all labels are disabled. In particular, $fairExec(etr)$ for a finite trace *etr* implies that all threads have terminated.

Lemma 2.2. $\forall etr, mtr.\ etr \hat{\preccurlyeq}_{\xi^{\lambda}_{\mathtt{OM}}} mtr \wedge fairExec(etr) \implies oblsFair(mtr)$.

Lemma 2.3. *If both* $\prec_{\mathsf{L}}$ *and* $\prec_{\mathsf{D}}$ *are well-founded, then* $\forall mtr.\ oblsFair(mtr) \implies \mathsf{finite}(mtr)$

3 Program Logic

As explained in §2, our methodology to prove fair termination is to find a refinement between fair program execution traces and traces of OM which are guaranteed to terminate by Lemma 2.3. To this end, we have developed the Lawyer program logic (Hoare logic) on top of Trillium's [Timany et al. 2024] program logic. The program logic of Trillium can be instantiated with an arbitrary LTS model, which we here fix to be OM. Fixing the LTS to be OM allows us to reflect the state and transitions of OM into the program logic, which we can then use to derive the rules of the Lawyer program logic. By doing so, the Lawyer program logic provides reasoning principles to support modular proofs that programs refine the OM model.

In Lawyer, *Hoare triples* are written as $\{P\}\ e\ \{\Phi\}^{\tau}_{\mathcal{E}}$. Intuitively, $\{P\}\ e\ \{\Phi\}^{\tau}_{\mathcal{E}}$ means that, given that the precondition P holds, the program e is safe to run (*i.e.* without getting stuck) under thread τ, every step of e is matched by a transition in OM (hence establishing the refinement), and that whenever e reduces to a value v, the postcondition $\Phi(v)$ holds. Furthermore, in the proof of $\{P\}\ e\ \{\Phi\}^{\tau}_{\mathcal{E}}$ only invariants whose names appear in the mask $\mathcal{E}$ can be accessed. Whenever all invariants are accessible, *i.e.*, the mask $\mathcal{E}$ is $\top$, the set of invariant names, we omit it.

In the rest of this section, we will explain the Lawyer program logic by working through the verification of the `nondet` example in Figure 7 (Page 10). As we do so, we will also introduce concepts of the higher-order concurrent separation logic Iris that are used to formalize Trillium and Lawyer. To state and prove the specification of `nondet`, we assume that OM is instantiated so that:

[2] That is, either they are both finite and have the same length, or they are both infinite.

Ownership of the resource …	implies … for δ, the current state of OM	persistent
bar (π : Phase) (d : Degree)	$\exists \pi_0.\ \{1 \cdot (\pi_0, d)\} \subseteq \delta.\mathtt{fuel} \land \pi_0 \preceq_{\mathsf{P}} \pi$	no
sgn (s : SignalId) (ℓ : Level) (b : **B**)	$\delta.\mathtt{sigs}[s] = (\ell, b)$	no
$\mathsf{sgn}_\exists$ (s : SignalId) (ℓ : Level)	$\exists b.\ \delta.\mathtt{sigs}[s] = (\ell, b)$	yes
obls (τ : Thread) (R : $FinSet$(SignalId))	$\delta.\mathtt{obls}[\tau] = R$	no
ep (s : SignalId) (π : Phase) (d : Degree)	$\exists \pi_0.\ (s, \pi_0, d) \in \delta.\mathtt{eps} \land \pi_0 \preceq_{\mathsf{P}} \pi$	yes
$\mathsf{ph}_{(q:\mathbb{Q}) \in (0..1]}$ (τ : Thread) (π : Phase)	$\delta.\mathtt{phs}[\tau] = \pi$	no
elb ($n : \mathbb{N}$)	$n \le \delta.\mathtt{eb}$	yes

Abbreviations:

$$\mathsf{ph}\ \tau\ \pi \triangleq \mathsf{ph}_1\ \tau\ \pi \qquad L \lhd \mathsf{sgn}\ R \triangleq \mathop{\Large *}_{s \in R} \exists \ell_s.\ (\mathsf{sgn}_\exists\ s\ \ell_s \land \bigwedge_{\ell \in L} \ell \lhd \ell_s) \quad \text{for} \quad \lhd \in \{\prec_{\mathsf{L}}, \preceq_{\mathsf{L}}\}$$

Fig. 9. Resources of the Lawyer logic

- There are at least two fuel degrees d_0, d_1 : Degree such that $d_0 \prec_{\mathsf{D}} d_1$.
- There is at least one obligation level ℓ_0 : Level.
- Every OM transition allows at least 5 small transitions, *i.e.* $5 \le \mathsf{SB}$.

How Modular are Lawyer Specifications? Note that the restrictions above *do not* require to choose either the exact Degree and Level orders or value of SB upfront. This allows to compose the modules' specifications: the restrictions placed by each of them are combined appropriately to obtain a specification for the composition. For example, §4.4 shows how the OM parameters' restrictions placed by an abstract lock module are used to determine the restrictions for a lock client program.

With that, we prove the following specification for an arbitrary thread τ and phase π:

$$\left\{\mathsf{obls}\ \tau\ \emptyset * \mathsf{ph}\ \tau\ \pi * \mathsf{bar}\ \pi\ d_1 * 13 \cdot \mathsf{bar}\ \pi\ d_0\right\} \mathtt{nondet}\ ()\ \left\{n.\ \mathsf{elb}\ n * \exists \pi' \succeq_{\mathsf{P}} \pi.\ \mathsf{obls}\ \tau\ \emptyset * \mathsf{ph}\ \tau\ \pi'\right\}^{\tau} \qquad \text{(nondet-spec)}$$

Intuitively, the precondition states (see Figure 9) that `nondet` can be called when the state δ of OM satisfies the following properties: thread τ holds no obligations and has phase π assigned to it, and that phase π has at least 1 barrel of d_1 and 13 barrels of d_0. Here, $*$ is the separating conjunction of separation logic, like ordinary conjunction $P * Q$ holds when both P and Q hold, but furthermore it requires the resources to be disjoint. To see this, note that $13 \cdot \mathsf{bar}\ \pi\ d_0$ in the spec above is a shorthand for $\mathsf{bar}\ \pi\ d_0 * \cdots * \mathsf{bar}\ \pi\ d_0$, 13 times, which means we have 13 individual barrels, because they are disjoint. Now, if the precondition just described holds, we know that `nondet` is safe to run, it refines OM, and that when it terminates, it returns $n \in \mathbb{N}$ and ends in some state δ' of OM such that the postcondition holds. The postcondition intuitively states that in the state δ' the exchange bound is at least n, written as the proposition $\mathsf{elb}\ n$, and that the thread τ still has no obligations, but its phase is now increased to π'. Having $\mathsf{elb}\ n$ in the postcondition lets us use the return value n of `nondet` for fuel exchanges, as we did while verifying Figure 3.

Intuitively, `nondet` terminates because the main thread sets the flag telling the forked thread to stop incrementing the counter. As explained in §2, the OM model represents this idea using signals which indicate that some other thread has the obligation to set the flag. Thus, to establish the spec (nondet-spec) above, we need to have a signal corresponding to the boolean value referenced by the flag. This is formally captured by the following logical *invariant*:

$$\mathsf{nondetInv}(cnt, flag, s_f) \triangleq \boxed{\exists c : \mathbb{N}, f : B.\ flag \mapsto f * cnt \mapsto c * \mathsf{sgn}\ s_f\ l_0\ f * \mathsf{elb}\ c}^{\mathcal{N}_{\mathsf{ND}}} \qquad \text{(nondet-inv)}$$

$$\textsc{OU-eb-0}\quad \mathsf{OU}\,\{\mathsf{elb}\,0\}$$

$$\textsc{OU-exchanges-fuel}\quad \frac{\mathsf{bar}\,\pi\,d \qquad \mathsf{elb}\,b \qquad d' \prec_{\mathsf{D}} d}{\mathsf{OU}\,\{b \cdot \mathsf{bar}\,\pi\,d'\}}$$

$$\textsc{OU-creates-signal}\quad \frac{\mathsf{obls}\,\tau\,R}{\mathsf{OU}\,\{\exists s \notin R.\ \mathsf{sgn}\,s\,\ell\,\mathsf{False} * \mathsf{obls}\,\tau\,R \cup \{s\}\}}$$

$$\textsc{OU-sets-signal}\quad \frac{\mathsf{obls}\,\tau\,R \qquad s \in R \qquad \mathsf{sgn}\,s\,\ell\,(b : \mathbf{B})}{\mathsf{OU}\,\{\mathsf{sgn}\,s\,\ell\,\mathsf{True} * \mathsf{obls}\,\tau\,R \setminus \{s\}\}}$$

$$\textsc{OU-creates-ep}\quad \frac{\mathsf{bar}\,\pi\,d \qquad \mathsf{sgn}_{\exists}\,s\,\ell \qquad \mathsf{ph}_q\,\tau\,\pi \qquad d' \prec_{\mathsf{D}} d}{\mathsf{OU}\,\{\mathsf{ep}\,s\,\pi\,d' * \mathsf{ph}_q\,\tau\,\pi\}}$$

$$\textsc{OU-increases-eb}\quad \frac{\mathsf{elb}\,b}{\mathsf{OU}\,\{\mathsf{elb}\,(b+1)\}}$$

$$\textsc{OU-expects-ep}\quad \frac{\mathsf{ep}\,s\,\pi\,d \qquad \mathsf{sgn}\,s\,\ell\,\mathsf{False} \qquad \mathsf{obls}\,\tau\,R \qquad \{\ell\} \prec_{\mathsf{L}} \mathsf{sgn}\,R \qquad \mathsf{ph}_q\,\tau\,\pi}{\mathsf{OU}\,\{\mathsf{bar}\,\pi\,d * \mathsf{sgn}\,s\,\ell\,\mathsf{False} * \mathsf{obls}\,\tau\,R * \mathsf{ph}_q\,\tau\,\pi\}}$$

Fig. 10. Rules for the obligation update modality OU

which, in addition to requiring the signal, also requires that the fuel exchange bound must be at least as large as the value stored in the counter. Here, $l \mapsto v$ is the separation logic points-to proposition asserting exclusive ownership of the memory location l, and at the same time asserting its value to be v. Moreover, the proposition $\boxed{P}^{\mathcal{N}}$ asserts that P must always hold. The name of the invariant $\mathcal{N}$ is used for accounting purposes, ensuring that invariants are not accessed multiple times in a nested fashion, which is unsound in general. For example, in the case of the invariant above, accessing it twice in a nested fashion would give us two points-to propositions for the same location *flag*. Invariants are forced to always hold once they are established by only allowing them to be accessed for the duration of atomic operations:[3]

$$\textsc{inv-open-simpl}\quad \{R * P\}\ e\ \{Q * P\}^{\tau}_{\mathcal{E}\setminus\mathcal{N}} \wedge e \text{ is atomic} \wedge \mathcal{N} \subseteq \mathcal{E} \vdash \left\{R * \boxed{P}^{\mathcal{N}}\right\}\ e\ \{Q\}^{\tau}_{\mathcal{E}}$$

We also remark that invariants are *persistent*. Formally, a proposition is persistent if $\Box\, P \dashv\vdash P$, where $\Box$ is the so-called persistently modality and $\dashv\vdash$ is logical equivalence. Intuitively, $\Box\, P$ holds if P holds, and furthermore, P only expresses knowledge without asserting any exclusive ownership. Thus, $\Box\, P \vdash P$, $\Box\, P \vdash \Box\Box\, P$, and crucially, persistent propositions are duplicable, *i.e.*, $\Box\, P \dashv\vdash \Box\, P * \Box\, P$. Invariants being duplicable implies that they can be freely duplicated and shared with other threads.

3.1 Establishing the Invariant

After the first two lines of the `nondet` example (Figure 7), we obtain $\mathit{flag} \mapsto \texttt{false} * \mathit{cnt} \mapsto 0$. Thus, to establish the invariant (`nondet-inv`) we also need to obtain $\mathsf{sgn}\,s_f\,l_0\,\mathsf{False} * \mathsf{elb}\,0$. To this end, we need to take transitions in `OM`, in particular, we need to take OMS-CREATES-SIGNAL to obtain a signal (and a corresponding obligation). To this end, the Lawyer program logic features bespoke modalities and rules for reflecting (taking) transitions in `OM`. Following Iris and Trillium, while we give specifications and their proofs using Hoare triples, these are not primitive concepts of the logic — they are defined in terms of weakest preconditions: $\{P\}\ e\ \{v.\, Q\}^{\tau}_{\mathcal{E}} \triangleq \Box\left(P \twoheadrightarrow \mathsf{wp}^{\tau}_{\mathcal{E}}\ e\ \{v.\, Q\}\right)$ — the "persistently" modality $\Box$ ensures that the precondition captures *all* that is required for running the program, thus ensuring that Hoare triples are "self-contained" and hence duplicable.

[3]For simplicity, we omit the later modality in this paper; see Jung et al. [2018] for details. For those familiar with Iris, we remark that all invariants we formally state in the paper are timeless anyway.

Following Hinrichsen et al. [2024], we decompose weakest preconditions into two parts: a *single-step* weakest precondition [Liu et al. 2023] followed by a *model update* modality:

TRILLIUM-STEP-NVAL-SIMPL
$$\mathsf{sswp}\ e\ \left\{e'.\ \mathsf{MU}^\tau\{\mathsf{wp}^\tau\ e'\ \{\Phi\}\}\right\} \wedge e \notin \mathit{Val} \vdash \mathsf{wp}^\tau\ e\ \{\Phi\}$$

Single-step weakest preconditions are like weakest preconditions except they demand their post-condition to hold after one step of the program — note how the binder in the postcondition binds an expression e' and not a value like ordinary weakest preconditions do. The model update modality intuitively embeds model steps $\xrightarrow[\mathtt{OM}]{\tau}$. That is, $\mathsf{MU}^\tau\{P\}$ holds if P holds after a single step of $\xrightarrow[\mathtt{OM}]{\tau}$ is taken. Crucially, the model update modality never invalidates already established facts, meaning that it satisfies "framing": $P * \mathsf{MU}^\tau\{Q\} \vdash \mathsf{MU}^\tau\{P * Q\}$. Just as `OM` transitions are broken into multiple *small transitions*, the model update modality allows taking multiple small transitions through a bespoke *obligations update* modality OU:

MU-OM-SIMPL
$$\mathsf{OU}^i\ \{P * \mathsf{ph}\ \tau\ \pi * \mathsf{bar}\ \pi\ d\} \wedge i \le \mathsf{SB} \vdash \mathsf{MU}^\tau\{P * \mathsf{ph}\ \tau\ \pi\}$$

where $\mathsf{OU}^i\ \{P\}$ is i times repeated obligation update modalities: $\mathsf{OU}^0\ \{P\} \triangleq R$ and $\mathsf{OU}^{n+1}\ \{P\} \triangleq \mathsf{OU}\ \{\mathsf{OU}^n\ \{P\}\}$. Note how this rule requires a barrel of fuel (at any degree d) — this corresponds to the fact that any `OM` transition burns fuel. (This rule is not a primitive rule of the logic, as we will discuss in §4.1.) The rules for deriving obligation updates are given in Figure 10. Using these rules, we can derive the following

$$\mathsf{obls}\ \tau\ \emptyset * \mathsf{ph}\ \tau\ \pi * \mathsf{bar}\ \pi\ d_1 \vdash \mathsf{OU}\ \{\mathsf{OU}\ \{\exists s_f.\ \mathsf{obls}\ \tau\ \{s_f\} * \mathsf{ph}\ \tau\ \pi * \mathsf{sgn}\ s_f\ \ell_0\ \mathsf{False} * \mathsf{ep}\ s_f\ \pi\ d_0\}\} \tag{nondet-ou-upd}$$

where $\mathsf{ep}\ s\ \pi\ d$ asserts existence of an expectation permission for signal s, allowing to generate fuel at degree d and phase of at least π. This is passed along with the invariant to the forkee running `incr_loop` so that it, together with $\mathsf{sgn}\ s_f\ \ell_0\ \mathsf{False}$ in the invariant, can be used to generate fuel while waiting for the signal. (Note that $\mathsf{sgn}_\exists\ s\ \ell$ is the persistent knowledge that the signal exists for which we have $\mathsf{sgn}\ s_f\ \ell\ b \vdash \mathsf{sgn}_\exists\ s_f\ \ell$.) We use the obligation update (nondet-ou-upd) above after allocating the flag, and then use the rule OU-EB-0 to obtain $\mathsf{elb}\ 0$ after allocating the counter.

This establishes all the resources necessary for the invariant (nondet-inv) and leaves us with $\mathsf{obls}\ \tau\ \{s_f\} * \mathsf{ep}\ s_f\ \pi\ d_0$. At this point, we use the rule MU-INV-ALLOC-SIMPL to create the invariant (nondet-inv).

MU-INV-ALLOC-SIMPL
$$\mathsf{MU}^\tau\{P * Q\} \vdash \mathsf{MU}^\tau\{P * \boxed{Q}^{\mathcal{N}}\}$$

3.2 Verifying `nondet`'s Main Thread

The proof of the `nondet` specification (nondet-spec) is sketched in Figure 11. The annotations in curly braces are propositions that hold between steps; persistent propositions are marked with □ and are not explicitly included in subsequent annotations. Pure reductions, such as the one on line 2, are normally verified simply by burning a barrel of fuel. In other words, we prove the triple on lines 1-3 by framing all resources, except for the phase and one barrel of d_0 which we use together with MU-OM-SIMPL with $i \triangleq 0$. Establishing resources after flag allocation (line 6) and counter allocation (line 12), as well as the creation of the invariant (nondet-inv), have been discussed above.

To verify the forking on line 16, we use the following rule that corresponds to a forking transition in `OM` without either preceding small transitions or delegating obligations to the forked thread:

TRILLIUM-STEP-FORK-SIMPL
$$\frac{\forall \tau'.\ (\exists \pi_1, \pi_2.\ \mathsf{ph}\ \tau\ \pi_1 * \mathsf{ph}\ \tau'\ \pi_2 * \mathsf{obls}\ \tau'\ \emptyset * \pi \prec_{\mathrm{P}} \pi_1 \wedge \pi \prec_{\mathrm{P}} \pi_2) \twoheadrightarrow (\mathsf{wp}^{\tau'}\ e\ \{\mathsf{obls}\ \tau'\ \emptyset\} * P)}{\mathsf{ph}\ \tau\ \pi * \mathsf{bar}\ \pi\ (d : \mathsf{Degree}) \vdash \mathsf{wp}^\tau\ \mathbf{fork}\ (e)\ \{P\}}$$

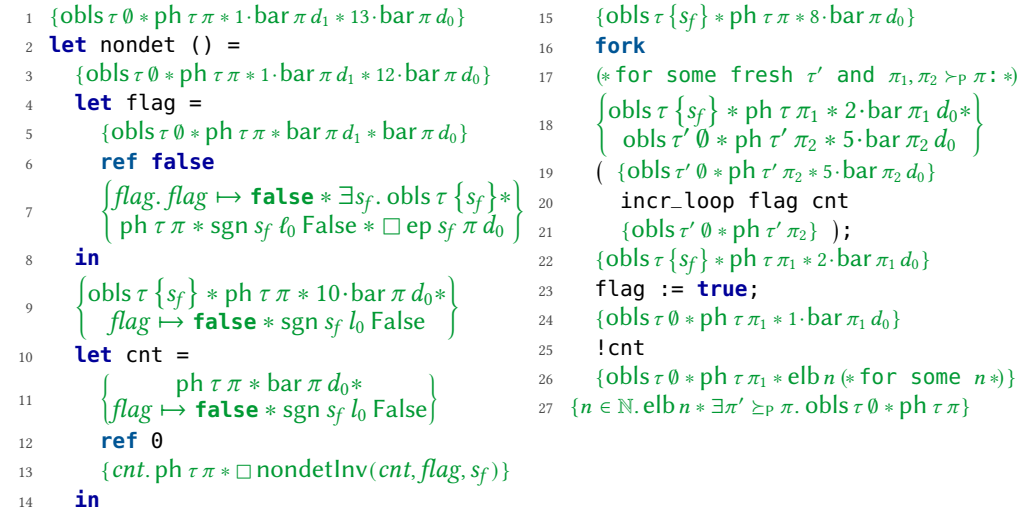


Fig. 11. Proof of `nondet` specification

This is a special derived rule for the `nondet` example; the general version is shown in [Namakonov et al. 2026b]. We split the available resources to verify the two threads separately. The verification of the forked thread is described below in §3.3. Notice that postconditions of both threads in Figure 11 require the set of obligations to be empty: $\mathsf{obls}\,\tau\,\emptyset$ for the main thread and $\mathsf{obls}\,\tau'\,\emptyset$ for the forked thread. Lawyer requires to prove that each thread has no obligations upon termination. Otherwise, a thread can terminate while holding obligations that other threads might depend on, causing them to wait indefinitely. The main thread fulfills its obligation by setting `flag` on line 23 for which we prove the following, which requires accessing the invariant using the rule INV-OPEN-SIMPL:

$$\{\mathsf{obls}\,\tau\,\{s_f\} * \mathsf{ph}\,\tau\,\pi_1 * \mathsf{bar}\,\pi_1\,d_0 * \mathsf{nondetInv}(cnt, flag, s_f)\}\ \texttt{flag := true}\ \{\mathsf{obls}\,\tau\,\emptyset * \mathsf{ph}\,\tau\,\pi_1\}^{\tau}$$

To prove this we use the rule MU-OM-SIMPL with $i \triangleq 1$ and OU-SETS-SIGNAL to derive the following:

$$\mathsf{sgn}\,s_f\,l_0\,(b : B) * \mathsf{obls}\,\tau\,\{s_f\} \vdash \mathsf{MU}^{\tau}\{\mathsf{sgn}\,s_f\,l_0\,\mathsf{True} * \mathsf{obls}\,\tau\,\emptyset\}$$

Finally, we read the counter on line 25, again accessing the invariant using the rule INV-OPEN-SIMPL to obtain the points-to and obtain the $\mathsf{elb}\,n$ resource. The latter we can retain after we are done accessing the invariant because it is persistent. Thus, for reading the counter we show:

$$\{\mathsf{ph}\,\tau\,\pi_1 * \mathsf{bar}\,\pi_1\,d_0 * \mathsf{nondetInv}(cnt, flag, s_f)\}\ \texttt{!cnt}\ \{n \in \mathbb{N}.\mathsf{ph}\,\tau\,\pi_1 * \mathsf{bar}\,\pi_1\,d_0 * \mathsf{elb}\,n\}^{\tau}$$

3.3 Verifying `incr_loop`

Figure 12 sketches the proof of the `incr_loop` specification used in Figure 11 (lines 19-21). Note the recursive call on line 11. Following the standard way to verify recursive functions [Birkedal and Bizjak 2017], we assume that the given spec holds for the recursive call while we show that the body of the recursive function satisfies the given spec. Thus, upon the recursive call we need to prove that the precondition on line 1, which includes fuel ($5{\cdot}\mathsf{bar}\,\pi\,d_0$), holds again. However, going through the lines before the recursive call, we have burned some of the initial fuel we started with in the precondition. Therefore, fuel has to be replenished before the recursive call. We do this at the point when we read the flag on line 4. There, after opening the invariant, executing the

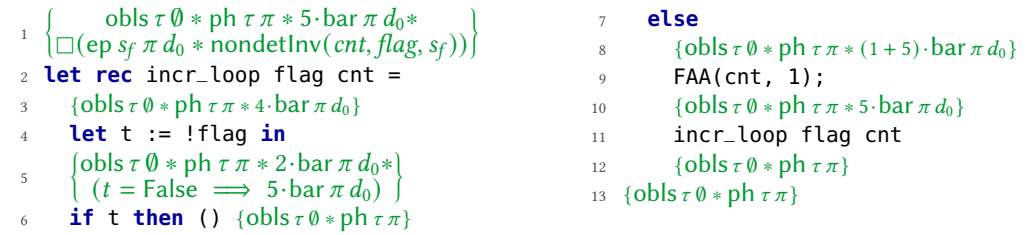


Fig. 12. Proof sketch for `incr_loop`

physical step and framing the irrelevant resources, we prove the following model update using MU-OM-SIMPL choosing i depending on b:

$$\frac{\mathsf{obls}\,\tau\,\emptyset * \mathsf{ph}\,\tau\,\pi * \mathsf{bar}\,\pi\,d_0 * \mathsf{sgn}\,s_f\,l_0\,b * \mathsf{ep}\,s_f\,\pi\,d_0}{\mathsf{MU}^\tau\{\mathsf{obls}\,\tau\,\emptyset * \mathsf{ph}\,\tau\,\pi * \mathsf{sgn}\,s_f\,l_0\,b * (b = \mathsf{False} \implies 5{\cdot}\mathsf{bar}\,\pi\,d_0)\}}$$

If $b = \mathsf{True}$, we take i to be 0; this case is trivial. If $b = \mathsf{False}$, on the other hand, we take i to be 5 and then apply OU-EXPECTS-EP 5 times; this is why we placed a restriction $5 \leq \mathsf{SB}$ at the beginning of this section. The rest of the proof also proceeds by case analysis. If the flag has been set, `incr_loop` terminates. Otherwise, we open the invariant again to increment the counter and update the exchange bound resource using the rule OU-INCREASES-EB. After this step, we have again all the resources comprising the precondition of `incr_loop` and thus can make the recursive call.

4 Case Study: Modular Verification of a Fair Lock Client

In this section, we demonstrate the expressivity of Lawyer and its program logic by proving fair termination of a program, given in Figure 13, consisting of two threads, called `left` and `right`, that work on a shared memory location `flag` protected by a ticket lock. The key point here is again modularity: the two threads are verified relative to a *specification* of fair locks, which we separately show to be satisfied by the ticket lock implementation. In more detail, the `left` thread writes to the shared location `flag`, and the `right` thread busy-waits for this write. Intuitively, the `client` terminates with any *fair* lock implementation, *i.e.*, one that guarantees that every lock acquisition eventually terminates, assuming that the lock is always eventually released. In other words, the proof of `client`'s termination need not depend on the implementation details of the lock. This is what we formally show by giving a modular proof as described above. However, the termination of the `client` program indeed depends on the *fairness* of the lock. If we had taken, *e.g.*, a spin lock, a pathological *but fair* scheduling in which every attempt to acquire the lock by the `left` thread is disrupted by the acquire operation in the `right` thread would lead to non-termination.

From a high-level point of view, the behavior of the `client` program here is similar to that of the `nondet` example: one thread waits for the other to set the flag. Naturally, the high-level idea of verification of the `client` program is also similar to that of the `nondet` program. However, the presence of the lock adds another source of blocking behavior — each acquire in either thread may need to wait for the other thread to release the lock.

The essence of the reason why a program like the one in Figure 13 terminates is that every lock acquisition is followed by a release. Lawyer's mechanism for specifying such requirements, as we saw in §2 and §3, is using obligations. Thus, we need the acquire operation to produce an obligation that can be fulfilled when the lock is released.

To verify this program, imagine first that the ticket lock implementation is inlined into the client program to set aside modularity concerns for the moment. In that case, when either thread acquires

```
1  let newLock () = (ref(0), ref(0))
2
3  let release (ow, _) = ow := !ow + 1
4
5  let acquire (ow, tk) =
6    let t = FAA(tk, 1) in
7    let rec wait () =
8      let o = !ow in
9      if (t == o) then () else wait ()
10   in wait ()
11
12 let left flag lk =
13   acquire lk; flag := true; release lk
14 let rec right flag lk =
15   let rec wait () =
16     acquire lk;
17     let c = !flag in
18     release lk;
19     if c then () else wait ()
20   in wait ()
21
22 let client () =
23   let flag = ref false in
24   let lk = newLock () in
25   fork (right flag lk);
26   left flag lk
```

Fig. 13. Client using the fair lock and ticket lock implementation

$$\frac{}{\mathrm{BOU}_n^{\mathcal{E}}\,\{P\}}\ \text{BOU-INTRO}\ P \qquad \frac{}{\mathrm{BOU}_n^{\mathcal{E}}\,\{P\} \twoheadrightarrow \mathrm{BOU}_n^{\mathcal{E}}\,\{Q\}}\ \text{BOU-WAND}\ P \twoheadrightarrow Q$$

$$\frac{\mathrm{OU}\,\{\mathrm{BOU}_n^{\mathcal{E}}\,\{P\}\}}{\mathrm{BOU}_{n+1}^{\mathcal{E}}\,\{P\}}\ \text{BOU-OU} \qquad \frac{\mathrm{BOU}_n^{\mathcal{E}}\,\{\mathrm{BOU}_m^{\mathcal{E}}\,\{P\}\}}{\mathrm{BOU}_{n+m}^{\mathcal{E}}\,\{P\}}\ \text{BOU-SPLIT}$$

$$\frac{P_1 \twoheadrightarrow P_2 \qquad n_1 \le n_2 \qquad \mathcal{E}_1 \subseteq \mathcal{E}_2 \qquad \mathrm{BOU}_{n_1}^{\mathcal{E}_1}\,\{P_1\}}{\mathrm{BOU}_{n_2}^{\mathcal{E}_2}\,\{P_2\}}\ \text{BOU-WEAKEN} \qquad \frac{{}_{\mathcal{E}}\Rrightarrow_{\emptyset} \mathrm{BOU}_n^{\emptyset}\,\{{}_{\emptyset}\Rrightarrow_{\mathcal{E}} P\}}{\mathrm{BOU}_n^{\mathcal{E}}\,\{P\}}\ \text{BOU-INVS}$$

Fig. 14. Selected rules for BOU

the lock, *i.e.*, when FAA at line 6 succeeds, we can assign that thread an obligation to release the lock. This obligation is fulfilled upon the write at line 3. On the other hand, when a thread fails to acquire the lock (*i.e.*, the check at line 9 fails), it knows that the lock owner is obligated to release it, which justifies the busy-waiting. As for the other blocking behavior, *i.e.*, waiting for the `flag` to be set, to justify the busy-waiting we follow the process from §3 and assign the left thread an obligation, which it fulfills upon setting the `flag`.

The idea behind modular verification of termination of `client` is to give a specification for the fair locks which, in addition to the core property of locking mechanisms, mutual exclusion, also captures what we discussed above: upon lock acquisition, an obligation is assigned for releasing it.

In the rest of this section, we will first introduce a new modality (§4.1) that allows us to achieve better flexibility in modular specifications. Afterwards, in §4.2 we show how to use this modality to give the fair lock specification, and discuss how the ticket lock implementation satisfies it in §4.3. Finally, in §4.4 we briefly explain how to verify `client` on top of this fair lock specification.

4.1 Bounded Obligations Update Modality

Recall that the OU modality presented in §3 represents exactly one small `OM` transition. We now introduce a more general variant of this modality, called the "bounded obligations update" modality, $\mathrm{BOU}_n^{\mathcal{E}}\,\{P\}$, which intuitively expresses that P holds after taking up to n small transitions and using invariants in $\mathcal{E}$ — we call n the "size" of $\mathrm{BOU}_n^{\mathcal{E}}\,\{P\}$. Figure 14 shows the proof rules governing BOU. Among them, the rules BOU-INTRO and BOU-OU correspond to the intuition of "up to n small transitions". The rule MU-OM-NOFORK is the main rule that allows us to use this new modality in

$$\forall \tau, \pi, u, P.\ \left\{P * \mathcal{B}_{\mathsf{FL}}(u) \leq \mathsf{SB} * \mathsf{bar}\ \pi\ d_{\mathsf{FL}} * \mathsf{ph}\ \tau\ \pi\right\}\ \mathtt{newLock}\ ()\ \left\{lk.\ \mathsf{FairLock}^{lk}(u, P) * \mathsf{ph}\ \tau\ \pi\right\}^{\tau} \qquad (\textit{newlockSpec})$$

where

$$\begin{aligned}
\mathsf{FairLock}^{lk}(u, P) \triangleq\ & \exists\, \mathsf{locked} : \mathsf{SignalId} \to \mathit{iProp},\ \iota_{\mathsf{FL}} : \mathcal{N}. \\
& \square\ (\forall s_1, s_2.\ \mathsf{locked}(s_1) * \mathsf{locked}(s_2) \mathrel{-\!\!*} \mathsf{False}) \wedge && (\text{lockedExcl}) \\
& \square\ \left(\forall s_r.\ \mathsf{locked}(s_r) \mathrel{-\!\!*} \exists l_r \in L_{\mathsf{FL}}.\ \mathsf{sgn}_{\exists}\ s_r\ l_r\right) \wedge && (\textit{lockedLvl}) \\
& \left(\begin{array}{l}\forall \tau, \pi, O. \\ \left\{\mathsf{bar}\ \pi\ d_{\mathsf{FL}} * \mathsf{ph}\ \tau\ \pi * \mathsf{obls}\ \tau\ O * L_{\mathsf{FL}} \prec_{\mathsf{L}} \mathsf{sgn}\ O\right\} \\ \quad \mathtt{acquire}\ lk \\ \left\{\exists s_r \notin O.\ P * \mathsf{locked}(s_r) * \mathsf{ph}\ \tau\ \pi * \mathsf{obls}\ \tau\ (O \cup \{s_r\})\right\}^{\tau}\end{array}\right) \wedge && (\textit{acquireSpec}) \\
& \left(\begin{array}{l}\forall \tau, \pi, O, s_r, Q. \\ \left\{\begin{array}{l}\mathsf{locked}(s_r) * \mathsf{bar}\ \pi\ d_{\mathsf{FL}} * \mathsf{ph}\ \tau\ \pi * \mathsf{obls}\ \tau\ (O \cup \{s_r\}) * \\ \left(\forall q.\ \mathsf{ph}_q\ \tau\ \pi \mathrel{-\!\!*} \mathsf{obls}\ \tau\ O \mathrel{-\!\!*} \mathsf{BOU}_u^{\top \setminus \{\iota_{\mathsf{FL}}\}}\ \{P * (\mathsf{ph}_{1-q}\ \tau\ \pi \mathrel{-\!\!*} Q)\}\right)\end{array}\right\} \\ \quad \mathtt{release}\ lk \\ \{Q\}^{\tau}\end{array}\right) && (\textit{releaseSpec})
\end{aligned}$$

Fig. 15. Fair lock specification (parameterized by $L_{\mathsf{FL}} : \mathit{FinSet}(\mathsf{Level})$, $d_{\mathsf{FL}} : \mathsf{Degree}$ and $\mathcal{B}_{\mathsf{FL}} : \mathbb{N} \to \mathbb{N}$)

proofs when performing a model update:

$$\frac{}{\mathsf{BOU}_{\mathsf{SB}}^{\mathcal{E}}\ \{P * \mathsf{ph}_q\ \tau\ \pi * \exists d.\ \mathsf{bar}\ \pi\ d\} \vdash \mathsf{MU}_{\mathcal{E}}^{\tau}\{P * \mathsf{ph}_q\ \tau\ \pi\}}\ \text{MU-OM-NOFORK}$$

In particular, the rule MU-OM-SIMPL we had shown earlier in §3 can be derived from MU-OM-NOFORK.

4.2 Fair Lock Specification

Our specification for a fair lock is given in Figure 15. Ignoring the parts highlighted in blue, one can recognize the pre- and post-conditions of the standard lock specification in higher-order concurrent separation logic [Birkedal and Bizjak 2017]. Indeed, in (*newlockSpec*) when the lock is created, the user provides P — the resources protected by the lock. In return, `newLock` establishes a persistent resource $\mathsf{FairLock}^{lk}(u, P)$ stating that the lock lk protects the resource P (we address the u parameter below). When `acquire` succeeds, as (*acquireSpec*) indicates, the user obtains P along with the exclusive token (lockedExcl) that ensures that no other thread holds the lock at the moment. Upon `release`, `release`'s specification, (*releaseSpec*) requires the user to return both P and this exclusive token; the postcondition of `release` in Birkedal and Bizjak [2017] is simply True. We emphasize that this lock specification in Figure 15, just like the one given by Birkedal and Bizjak [2017], is abstract: it hides the implementation by existentially quantifying over the predicate locked and the invariant name ι_{FL}.

The specification in Figure 15 is parameterized by a set of obligation levels, a degree and a function on natural numbers (described below). As the execution of any method consumes fuel, the fair lock specification mentions a fuel degree d_{FL}. We require a barrel of this degree is provided for every call

to a lock method, along with the calling thread's current phase, which is returned unchanged. The fair lock specification mandates the eventual release of an acquired lock by assigning the thread that acquires the lock a new obligation. This obligation can only be fulfilled by releasing the lock — `acquire` generates a signal s_r and locked(s_r) token, while the release operation fulfills s_r for which it receives locked(s_r). The level of the release obligation belongs to the set L_{FL}. The lock client can carry other obligations while acquiring the lock, but the levels of these other obligations must be higher than those in L_{FL}. Parameterization of Figure 15 allows imposing restrictions on *e.g.* obligations levels used by each individual lock instance created with `newLock`, which is needed to verify Figure 13 (see §4.4) and an example with multiple locks (provided in [Namakonov et al. 2026a]).

The reason for the release specification (*releaseSpec*) having an arbitrary postcondition and requiring a bounded update modality in the precondition is to make the specification sufficiently flexible for clients. In particular, the proof of termination of our example `client` in §4.4 relies on replenishing the right thread's fuel at the moment the lock is released. To enable this, the client of the lock can pick the postcondition Q to also include the newly created fuel and provide a BOU that replenishes the fuel.[4] (The use of fractional phase allows to support the ticket lock implementation, which handles phases in a non-trivial way. In practice, it does not restrict the user.) The size of the user-passed BOU in the (*releaseSpec*) is decided in (*newlockSpec*) when the lock is created. This value determines the *module bound* $\mathcal{B}_{\mathsf{FL}}(u)$ that limits the total size of BOU applied by the module. Finally, the user of the ticket lock is required to work with a version `OM` with an SB bound that satisfies $\mathcal{B}_{\mathsf{FL}}(u) \leq \mathsf{SB}$. The condition $\mathcal{B}_{\mathsf{FL}}(u) \leq \mathsf{SB}$ allows the fair lock's implementation to apply not just the passed BOU, but also to take all the small transitions that it internally needs.

4.3 Verifying the Ticket Lock Implementation

Here we discuss the proof that the ticket lock implementation satisfies the specification in Figure 15 (for any choice of parameters L_{FL}, d_{FL} and $\mathcal{B}_{\mathsf{FL}}$ satisfying a number of assumptions wrt. `OM`; see [Namakonov et al. 2026a]). For space reasons, we only briefly describe the proof from a high-level point of view. We refer the reader to the accompanying material [Namakonov et al. 2026a,b] for further details.

The proof proceeds in two stages. First, we derive the fair lock specification in Figure 15 from a more general lock specification employing so-called *total correctness logically-atomic triples (TCLAT)*. Subsequently, we prove that the ticket lock implementation satisfies the TCLAT fair lock specification. Again, the details for both steps are provided in [Namakonov et al. 2026a,b].

To describe what we mean by a TCLAT specification, we first recall that even for safety reasoning, modular reasoning about concurrent modules is challenging. Technically, one of the challenges is to ensure that a client's invariant can be appropriately updated when calling the module's operations. An invariant can only be accessed and updated during *physically* atomic operations; however, a module's operations are not physically atomic as they at the very least involve a method call. Logically-atomic triples [Jung et al. 2015; Rocha Pinto et al. 2014] are used for safety reasoning in state-of-the-art program logics to support modular reasoning by internalizing the well-known idea of linearizability [Herlihy and Wing 1990]. They allow one to pretend that a module's operation is atomic for all intents and purposes — hence the name *logically atomic triples* — by allowing invariants to be accessed and updated for the duration of the method call. In practice, behind the scenes, this pretense works by accessing and updating invariants at the so-called linearization point

[4]We remark that one could include a similar BOU in the `acquire` specification — it would be applied *after* the lock resource is obtained, but *before* the release obligation is produced; however, the client examples we consider require just one of these updates, and thus we choose to only include one for the `release` specification.

[Herlihy and Wing 1990] of the method in question. (For an introduction to such logically atomic specifications and proofs relying on them, see Birkedal and Bizjak [2017, Ch. 13].)

Similar challenges arise when verifying liveness properties of concurrent modules modularly. For example, the thread acquiring the lock must receive a release obligation exactly at the linearization point of the acquire method, *i.e.*, when the lock's observable state changes from "unlocked" to "locked". We capture such method's behavior with a "total correctness logically atomic triples" (TCLATs) — a version of logically atomic triples that allow, in addition to accessing invariants, to take transitions in the `OM` model at the linearization point of the method. Our TCLATs are inspired by a similar notion of triples from Sassy [Fasse and Jacobs 2024], but instead of using auxiliary code, we support taking `OM` transitions through BOU. This, as we explained in the Introduction, avoids all issues caused by ghost code instrumentation and significantly simplifies our Rocq mechanization.

The TCLAT specification style, as Sassy [Fasse and Jacobs 2024] shows, supports verifying a number of challenging examples, including lock handoffs and implementing nested locks, *e.g.*, cohort locks [Fasse and Jacobs 2024]. However, using the TCLAT-style lock specifications directly is cumbersome as it exposes details about the linearization point of methods that are not necessary for most clients; it is needed for more advanced clients, *e.g.*, verifying cohort locks on top of ticket locks [Fasse and Jacobs 2024]. Thus, our TCLAT-style specification of the fair lock is strong enough to verify such advanced case studies, but can also be used to derive the validity of the specs in Figure 15, which can be straightforwardly used to verify clients like that in Figure 13.

We finally remark that, similarly to logically atomic specifications, proving TCLAT-style specification of the ticket lock requires invariants to be impredicative, which in turn necessitates step-indexing. Lawyer inherits impredicative invariants from Iris. Predicative logics such as Lilo [Lee et al. 2025] work around this issue by sacrificing expressivity. Indeed, Lilo's specification of a ticket lock uses stratified propositions for the resources protected by the lock. Thus, no object referring to the lock itself can be protected by the lock. Furthermore, Lilo's specification of the ticket lock does not feature a logically atomic version.

4.4 Verification of the Lock Client Program

To state and prove the specification of the `client` function in Figure 13, we assume a lock implementation that satisfies the specification in Figure 15 for some d_{FL}, L_{FL} and $\mathcal{B}_{\mathsf{FL}}$. Moreover, we assume the following holds for the obligations model's parameters:

- besides d_{FL}, there are two other fuel degrees d_0, d_{CL} such that $d_0 \prec_{\mathsf{D}} d_{\mathsf{FL}} \prec_{\mathsf{D}} d_{\mathsf{CL}}$
- besides the set L_{FL} of obligation levels, there is a level l_f such that $\forall l \in L_{\mathsf{FL}}.\, l \prec_{\mathsf{L}} l_f$
- $\max(\mathcal{B}_{\mathsf{FL}}(4), 2) \leq \mathsf{SB}$

Note that these assumptions still don't fix the concrete choice for Level, Degree and SB and can be composed further with assumptions for other program modules, if any. In that way, our specifications stay modular.

The overall specification we give for the `client` function is as follows:

$$\left\{\mathsf{obls}\,\tau\,\emptyset * \mathsf{ph}\,\tau\,\pi * 3{\cdot}\mathsf{bar}\,\pi\,d_{\mathsf{CL}}\right\}\ \texttt{client}\ ()\ \left\{\exists \pi' \succeq_{\mathsf{P}} \pi.\ \mathsf{obls}\,\tau\,\emptyset * \mathsf{ph}\,\tau\,\pi'\right\}^{\tau}$$

Due to space reasons, we relegate the details of the proof to our accompanying technical appendix [Namakonov et al. 2026b].

5 Related Work

Lawyer is a higher-order impredicative concurrent program logic. It supports modular obligations-based reasoning about the termination of fine-grained concurrent programs in a proof assistant. The work closest to ours in expressivity and abstractions provided is Sassy [Fasse and Jacobs 2024],

which we have discussed in the Introduction and throughout the paper. Our model `OM` is inspired by Sassy's auxiliary code, inheriting its amenability to modular specification. However, unlike Lawyer, neither Sassy's soundness nor its case studies are mechanized in a proof assistant.

Non-Impredicative Modular Liveness Verification. Most program logics for liveness verification are either only first-order or only support predicative higher-order reasoning. As a result, these systems' support for modularity is limited. As Svendsen and Birkedal [2014] explain, truly modular separation logic specifications require *impredicativity*. Lilo [Lee et al. 2025] provides a mechanized program logic that abstracts the construction of thread-local simulation relations of [Lee et al. 2023]. They verify termination of a number of lock-based programs, as well as two stack implementations. To the best of our knowledge, Lawyer supports all of these examples. Moreover, Lilo supports verification of non-terminating programs. However, the usage of thread-local simulation restricts the kind of properties provable for them.[5] Lilo is a predicative higher-order logic in a way not completely unlike universes in proof assistants like Rocq, Agda, Lean, *etc.* This limits its support for modularity, *cf.* the discussion in §4.2. Moreover, Lilo only supports parallel composition instead of unstructured forking. It also does not address the verification of modules' initialization functions, *e.g.*, our `newLock` function in Figure 13, requiring to specify all modules' invariants in the initial state. Both Sassy and Lilo, as well as Lawyer, take inspiration from TaDA Live [D'Osualdo et al. 2019], which introduced the notions of "liveness invariants" and "impedance", which together allowed one to express a module's liveness dependencies on its environment, and to modularly prove termination. TaDA is a first-order logic and not mechanized. Lawyer overcomes some of the limitations of TaDA Live by supporting unstructured concurrency and scheduler non-determinism. However, the first-order setting of TaDA Live allows it to verify the lock-coupling set example, which is currently not supported by Lawyer because of its step-indexed nature; we discuss this limitation and a way to alleviate it in §6. Simuliris [Gäher et al. 2022] is built on top of the Iris framework but removes step-indexing, and thereby also higher-order ghost state, including impredicative invariants. Simuliris supports liveness reasoning through proving fair-termination-preserving refinement between programs. Yet, it does not support reasoning about blocking behavior. Lili [Liang and Feng 2016] verifies properties of blocking and non-blocking concurrent modules. Lili introduced the notions of "blocking" and "delay" (explained in §2) as two distinct types of interaction with the environment that affect a given module's termination. Lili is first-order and not mechanized. There is also work on non-program logic frameworks for liveness verification of first-order programs [Gu et al. 2015, 2018; Kim et al. 2017]. In particular, Kim et al. [2017] prove liveness of the MCS lock. However, it requires explicit trace-based reasoning about program semantics and the scheduler.

Non-Modular Impredicative Termination Verification. As explained in the Introduction, Trillium [Timany et al. 2024] works around the limitations of step-indexing by proving refinements to a specific class of abstract models. Trillium uses arbitrary abstract LTS models to verify a wide range of trace-level properties, including arbitrary safety and liveness properties, not just termination. This means that to use Trillium, the user must choose a concrete LTS model that captures the behavior of the program. Trillium does provide Fairis [Timany et al. 2024], a program logic that provides some degree of abstraction for working with an arbitrary model to prove termination; it essentially only builds in a rudimentary notion of fuel. Using Fairis requires the user to choose a model for the entire program. This precludes giving modular specifications to reusable modules as we did here for fair locks in §4. Tassarotti et al. [2017] showed that Iris can be used to reason about fair termination-preserving refinement between two programs, which they used for proving the

[5] For example, a program with two threads flipping a boolean flag in turn, waiting for each other to do so, is refined to a program where one thread repeatedly writes True, and another writes False. The fact that the flag is being flipped is missed.

correctness of a compiler. However, to achieve this, they had to extend Iris with linear propositions and, furthermore, had to restrict programs' non-determinism.

Impredicative Termination Verification of Coarse-Grained Concurrency. Some projects [Matsushita and Tsukada 2025; Spies et al. 2021] support modular termination verification in an impredicative setting, but they restrict the programs they can verify. Transfinite Iris [Spies et al. 2021] extended Iris's step-indexing to transfinite ordinals. This way, Transfinite Iris supports termination of *sequential* programs. However, it is unclear how one can extend this approach to support reasoning about the termination of blocking concurrent programs. Nola [Matsushita and Tsukada 2025] uses Iris' [Jung et al. 2015] total version of Hoare triples defined as an inductive type, ensuring that programs specified with such triples are terminating. Originally, the disadvantage of such triples was the lack of support for invariants. Nola's innovation is to change the semantic interpretation of invariants so that *non-impredicative* invariants can be accessed by total Hoare triples, while arbitrary invariants can be used in other contexts. This approach is limited to proving termination by meta-level induction, ruling out verification of blocking concurrency.

6 Conclusion, Discussion and Future Work

We have presented Lawyer, a higher-order impredicative program logic, which supports modular, obligations-based reasoning about termination of fine-grained concurrent programs. We have formalized both the meta-theory of Lawyer and the case studies in the Rocq proof assistant. To the best of our knowledge, Lawyer is the first impredicative higher-order concurrent separation logic that supports modular liveness reasoning formally in a proof assistant.

Verifying Properties Beyond Fair Termination. Verifying a program with Lawyer establishes a refinement between the program and our model OM. This allows to prove termination of closed programs under the assumption that the thread scheduler is fair. We note that it is possible to show termination even under unfair schedulers for programs that do not involve busy-waiting, by using specific instantiations of Level and Degree. For example, we have shown that the programs in Figure 4 and Figure 2 terminate even under an unfair scheduler. Moreover, we have also used a specific instantiation of OM to show the program in Figure 2 terminates within a time bound linear wrt. the value stored in location l initially. We refer the reader to our Rocq development and technical appendix [Namakonov et al. 2026a,b] for details. As a part of future work, we investigate the applicability of Lawyer to verifying lock-freedom of Figure 4 and wait-freedom of Figure 2.

Relative Image Finiteness Restriction. A limitation of Lawyer, which is inherited from Trillium, is that the refinement between the execution and the model traces must be relatively image-finite. This is the case for OM as we proved in Lemma 2.1. Proving it requires a number of restrictions on OM transitions.

- First, levels and degrees must be finite sets. Otherwise, model transitions that assign new signals and exchange fuel lead to infinite branching. Because of that, Lawyer as is cannot verify examples such as the lock-coupling set of TaDA Live, where each element of the set is assigned an obligation of a unique level and the set can grow arbitrarily large.
- For a similar reason, the eb component of the OM state limits the number of fuel barrels that can be obtained with a single exchange step. By using OMS-EXCHANGES-FUEL, we can verify examples like Figure 3 where this number grows arbitrarily, but incrementally. However, if `nondet` was implemented as an atomic primitive, this approach would not apply.
- Finally, we also need the SB parameter of OM to limit the number of small transitions within $\xrightarrow[\text{OM}]{}$. It limits how many signals can be created in a single step, and how many barrels of fuel

can be received by waiting on an unset signal. We are not aware of examples that would require lifting this restriction.

Future Work. As mentioned above, we aim to verify progress properties of independent modules (as opposed to verification of the entire program's termination). We also hope to extend Lawyer's approach to verify liveness of non-terminating programs. Finally, we aim to lift some of the aforementioned finite branching restrictions by employing more elaborate refinement relations, similar to how it is done in Trillium [Timany et al. 2024, the `nondet_start` example].

Data-Availability Statement

We provide the Rocq mechanization [Namakonov et al. 2026a] of all results presented in this paper, including definition of the Obligations model and the refinement relation $\xi^{\lambda}_{\mathsf{OM}}$, Lawyer program logic and its adequacy theorem, and proofs of termination of all the examples and case studies (Figure 13 and a number of others).

Acknowledgments

We would like to thank the anonymous reviewers for their valuable remarks and insightful comments which have improved the presentation of this work.

This work was supported in part by a Villum Investigator grant (VIL73403), Center for Basic Research in Program Verification (CPV, 25804), from Villum Fonden. This work was co-funded by the European Union (ERC, CHORDS, 101096090). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

This research was partially funded by the Research Fund KU Leuven and by the Cybersecurity Research Program Flanders.

References

Martín Abadi and Leslie Lamport. 1995. Conjoining specifications. *ACM Trans. Program. Lang. Syst.* 17, 3 (May 1995), 507–535. doi:10.1145/203095.201069

Andrew W. Appel. 2011. Verified software toolchain. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software* (Saarbrücken, Germany) *(ESOP'11/ETAPS'11)*. Springer-Verlag, Berlin, Heidelberg, 1–17.

Andrew W. Appel and David McAllester. 2001. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.* 23, 5 (sep 2001), 657–683. doi:10.1145/504709.504712

Lars Birkedal and Ales Bizjak. 2017. Lecture Notes on Iris: Higher-Order Concurrent Separation Logic. http://iris-project.org/tutorial-pdfs/iris-lecture-notes.pdf

Pontus Boström and Peter Müller. 2015. Modular Verification of Finite Blocking in Non-terminating Programs. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic (LIPIcs, Vol. 37)*, John Tang Boyland (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 639–663. doi:10.4230/LIPIcs.ECOOP.2015.639

Nachum Dershowitz and Zohar Manna. 1979. Proving termination with multiset orderings. *Commun. ACM* 22, 8 (Aug. 1979), 465–476. doi:10.1145/359138.359142

Emanuele D'Osualdo, Julian Sutherland, Azadeh Farzan, and Philippa Gardner. 2019. TaDA Live: Compositional Reasoning for Termination of Fine-grained Concurrent Programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 43 (2019), 1 – 134. doi:10.1145/347708

Justus Fasse and Bart Jacobs. 2024. A flexible specification approach for verifying total correctness of fine-grained concurrent modules. arXiv:2312.15379 [cs.PL] https://arxiv.org/abs/2312.15379

Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. 2022. Simuliris: a separation logic framework for verifying concurrent program optimizations. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–31. doi:10.1145/3498689

Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. *SIGPLAN Not.* 50, 1 (Jan. 2015), 595–608. doi:10.1145/2775051.2676975

Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified concurrent abstraction layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 646–661. doi:10.1145/3192366.3192381

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

Jonas Kastberg Hinrichsen, Léo Stefanesco, Lars Birkedal, and Amin Timany. 2024. Verifying Liveness Properties of Distributed Systems via Trace Refinement in Higher-Order Concurrent Separation Logic. (2024).

Aquinas Hobor, Robert Dockins, and Andrew W. Appel. 2010. A theory of indirection via approximation. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, Manuel V. Hermenegildo and Jens Palsberg (Eds.). ACM, 171–184. doi:10.1145/1706299.1706322

Bart Jacobs, Dragan Bosnacki, and Ruurd Kuiper. 2018. Modular Termination Verification of Single-Threaded and Multi-threaded Programs. *ACM Trans. Program. Lang. Syst.* 40, 3 (2018), 12:1–12:59. doi:10.1145/3210258

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6617)*, Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, 41–55. doi:10.1007/978-3-642-20398-5_4

Cliff B. Jones. 1983. Specification and Design of (Parallel) Programs. In *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, R. E. A. Mason (Ed.). North-Holland/IFIP, 321–332.

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

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 637–650. doi:10.1145/2676726.2676980

Jieung Kim, Vilhelm Sjöberg, Ronghui Gu, and Zhong Shao. 2017. Safety and Liveness of MCS Lock—Layer by Layer. In *Programming Languages and Systems*, Bor-Yuh Evan Chang (Ed.). Springer International Publishing, Cham, 273–297.

Naoki Kobayashi. 2006. A New Type System for Deadlock-Free Processes. In *CONCUR 2006 - Concurrency Theory, 17th International Conference, CONCUR 2006, Bonn, Germany, August 27-30, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4137)*, Christel Baier and Holger Hermanns (Eds.). Springer, 233–247. doi:10.1007/11817949_16

Dongjae Lee, Minki Cho, Jinwoo Kim, Soonwon Moon, Youngju Song, and Chung-Kil Hur. 2023. Fair Operational Semantics. *Proc. ACM Program. Lang.* 7, PLDI, Article 139 (June 2023), 24 pages. doi:10.1145/3591253

Dongjae Lee, Janggun Lee, Taeyoung Yoon, Minki Cho, Jeehoon Kang, and Chung-Kil Hur. 2025. Lilo: A Higher-Order, Relational Concurrent Separation Logic for Liveness. *Proc. ACM Program. Lang.* 9, OOPSLA1 (2025), 1267–1294. doi:10.1145/3720525

K. Rustan M. Leino, Peter Müller, and Jan Smans. 2010. Deadlock-Free Channels and Locks. In *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6012)*, Andrew D. Gordon (Ed.). Springer, 407–426. doi:10.1007/978-3-642-11957-6_22

Hongjin Liang and Xinyu Feng. 2016. A Program Logic for Concurrent Objects under Fair Scheduling. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 385–399. doi:10.1145/2837614.2837635

Zongyuan Liu, Sergei Stepanenko, Jean Pichon-Pharabod, Amin Timany, Aslan Askarov, and Lars Birkedal. 2023. VMSL: A Separation Logic for Mechanised Robust Safety of Virtual Machines Communicating above FF-A. *Proc. ACM Program. Lang.* 7, PLDI, Article 165 (June 2023), 25 pages. doi:10.1145/3591279

Yusuke Matsushita and Takeshi Tsukada. 2025. Nola: Later-Free Ghost State for Verifying Termination in Iris. *Proceedings of the ACM on Programming Languages* 9, PLDI (June 2025). doi:10.1145/3729250

Egor Namakonov, Justus Fasse, Bart Jacobs, Lars Birkedal, and Amin Timany. 2026a. Artifact for the "Lawyer: Modular Obligations- Based Liveness Reasoning in Higher-Order Impredicative Concurrent Separation Logic" paper of OOPSLA'26. doi:10.5281/zenodo.18457698

Egor Namakonov, Justus Fasse, Bart Jacobs, Lars Birkedal, and Amin Timany. 2026b. Technical appendix for the Lawyer paper. https://doi.org/10.5281/zenodo.18457698

Susan Owicki and David Gries. 1976. Verifying properties of parallel programs: an axiomatic approach. *Commun. ACM* 19, 5 (May 1976), 279–285. doi:10.1145/360051.360224

Tobias Reinhard and Bart Jacobs. 2021a. Ghost Signals: Verifying Termination of Busy Waiting. In *Computer Aided Verification*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer International Publishing, Cham, 27–50. doi:10.1007/978-3-030-81688-9_2

Tobias Reinhard and Bart Jacobs. 2021b. Ghost Signals: Verifying Termination of Busy-Waiting (Technical Report). https://people.cs.kuleuven.be/~tobias.reinhard/ghostSignals--TR.pdf

Pedro Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586*. Springer-Verlag, Berlin, Heidelberg, 207–231. doi:10.1007/978-3-662-44202-9_9

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. Mechanized verification of fine-grained concurrent programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 77–87. doi:10.1145/2737924.2737964

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

Kasper Svendsen and Lars Birkedal. 2014. Impredicative Concurrent Abstract Predicates. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings (Lecture Notes in Computer Science, Vol. 8410)*, Zhong Shao (Ed.). Springer, 149–168. doi:10.1007/978-3-642-54833-8_9

Joseph Tassarotti, Ralf Jung, and Robert Harper. 2017. A Higher-Order Logic for Concurrent Termination-Preserving Refinement. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10201)*, Hongseok Yang (Ed.). Springer, 909–936. doi:10.1007/978-3-662-54434-1_34

Amin Timany, Simon Oddershede Gregersen, Léo Stefanesco, Jonas Kastberg Hinrichsen, Léon Gondelman, Abel Nieto, and Lars Birkedal. 2024. Trillium: Higher-Order Concurrent and Distributed Separation Logic for Intensional Refinement. *Proc. ACM Program. Lang.* 8, POPL, Article 9 (jan 2024), 32 pages. doi:10.1145/3632851

Received 2025-10-10; accepted 2026-02-17