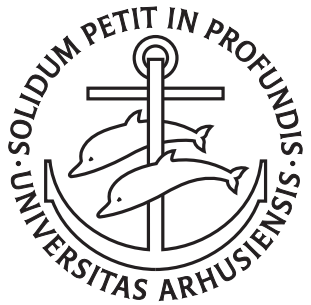
# Modular verification of liveness properties in higher-order concurrent languages

Egor Namakonov

PhD Dissertation




Department of Computer Science
Aarhus University
Denmark

# Modular verification of liveness properties in higher-order concurrent languages

A Dissertation
Presented to the Faculty of Natural Sciences
of Aarhus University
in Partial Fulfilment of the Requirements
for the PhD Degree

by
Egor Namakonov
October 27, 2026

# Abstract

Concurrent software is central to modern computing, powering systems such as operating systems, databases, and distributed applications. Ensuring its correctness is crucial, yet inherently difficult due to unpredictable interactions between concurrently executing threads. Among correctness guarantees, liveness properties—such as termination or non-blocking progress—are particularly challenging. Unlike safety properties, which deal with finite execution prefixes and can be tested for, liveness properties concern potentially infinite behaviors and therefore require a different approach.

Formal verification provides a way to establish such guarantees by modeling programs mathematically and reasoning about all their possible executions. However, verifying liveness properties of concurrent programs faces a major challenge of modularity. Ideally, one verifies complex programs by separately verifying their individual components and composing the results. While this is well-developed for safety, achieving similar modularity for liveness is difficult due to complex dependencies between components and their environment, including the scheduler. Capturing these dependencies in reusable specifications has been a non-trivial task. Moreover, the reasoning methods used in verifying safety of higher-order programs are not directly applicable to verifying liveness.

This thesis proposes a solution for modular verification of liveness properties in higher-order concurrent languages. The approach combines two key ideas. First, it builds on a refinement-based Trillium framework which enables reasoning about liveness in a higher-order setting. Second, it incorporates the abstractions of obligations (inspired by Sassy) to specify and verify liveness behavior compositionally. Together, these techniques yield the Lawyer program logic that supports modular reasoning about liveness, in a way analogous to established approaches for safety. The entire development is mechanized in the Rocq proof assistant on top of the Iris framework, ensuring a high degree of trust in the results.

I use the Lawyer logic for verifying two kinds of properties. First, I establish termination of closed programs under different scheduling assumptions. Second, I show that certain specifications imply wait-freedom of individual operations, even in unknown higher-order contexts.

Thus, the thesis established by this work is: **Refinement-based program logic, combined with obligations-based reasoning, allows one to modularly verify liveness properties of blocking and non-blocking higher-order concurrent programs**.

# Resumé

Samtidig software er central for moderne databehandling og driver systemer som operativsystemer, databaser og distribuerede applikationer. Det er afgørende at sikre dens korrekthed, men i sagens natur vanskeligt på grund af uforudsigelige interaktioner mellem samtidigt eksekverende tråde. Blandt korrekthedsgarantier er livlighedegenskaber - såsom afslutning eller ikke-blokerende fremskridt - særligt udfordrende. I modsætning til sikkerhedsegenskaber, der beskæftiger sig med endelige eksekveringspræfikser og kan testes for, vedrører livlighed-egenskaber potentielt uendelig adfærd og kræver derfor en anden tilgang.

Formel verifikation giver en måde at etablere sådanne garantier ved at modellere programmer matematisk og ræsonnere over alle mulige eksekveringer af dem. Verifikation af livlighedegenskaber for samtidige programmer står dog over for en stor udfordring med modularitet. Ideelt set verificerer man komplekse programmer ved separat at verificere deres individuelle komponenter og sammensætte resultaterne. Selvom dette er veludviklet for sikkerhed, er det vanskeligt at opnå lignende modularitet for livlighed på grund af komplekse afhængigheder mellem komponenter og deres miljø, inklusive planlægger. At indfange disse afhængigheder i genanvendelige specifikationer har været en ikke-triviel opgave. Desuden er de ræsonnementsmetoder, der anvendes til at verificere sikkerheden af højere ordens programmer, ikke direkte anvendelige til at verificere livlighed.

Denne afhandling foreslår en løsning til modulær verifikation af livlighed-egenskaber i højere ordens parallelle sprog. Tilgangen kombinerer to nøgleideer. For det første bygger den på et forfiningsbaseret Trillium-rammeværket, der muliggør ræsonnement om livlighed i en højere ordens kontekst. For det andet inkorporerer den abstraktioner af forpligtelser (inspireret af Sassy) for at specificere og verificere livlighed-adfærd kompositionelt. Sammen giver disse teknikker Lawyer-programlogikken, der understøtter modulær ræsonnement om livlighed på en måde, der er analog med etablerede tilgange til sikkerhed. Hele udviklingen er mekaniseret i Rocq-bevisassistenten oven på Iris-rammeværket, hvilket sikrer en høj grad af tillid til resultaterne.

Jeg bruger Lawyer-logikken til at verificere to slags egenskaber. For det første etablerer jeg afslutning af lukkede programmer under forskellige planlægningsantagelser. For det andet viser jeg, at visse specifikationer indebærer ventefrihed for individuelle operationer, selv i ukendte højere ordens kontekster. Tesen, der etableres i dette arbejde, er således: **Forfiningsbaseret programlogik, kombineret med forpligtelsesbaseret ræsonnement, gør det muligt modulært at verificere livlighed-**

**egenskaber for blokerende og ikke-blokerende samtidige programmer af højere orden**.

# Acknowledgments

*If it's hard to grasp*
*Follow time's path back*
*Who's holding your hand throughout?*
*Who's looking out through your eyes?*
**Karmanjakah, "Listening"**

First of all, this work would not be possible without the people who advised me. I am grateful to my advisors, Lars Birkedal and Amin Timany, for helping me to grow in every possible direction as a researcher, for being patient when the project was slowing down, and for trusting me when it had to be pivoted. I am also grateful to Anton Podkopaev, who introduced me to formal methods during our time at JetBrains Research, and to William Mansky, who has been my mentor over the past five years and also helped me with the first steps with Iris.

Besides that, more people contributed to this research. I learned a lot from the works of Bart Jacobs, with whom I had the honor to work, and who hosted me during my stay at KU Leuven. And I was lucky to meet Justus Fasse early in my program, who is not only the person who learned all the perils of liveness research like I did, but is also a great friend.

Then, I am grateful to those who made my life better outside of academia. First of all, thanks to my parents, Elena and Sergey, who made me a person daring enough to pursue a PhD abroad and have been supportive throughout my life. Then, thanks to my friends Alyosha, Nikita, Mikael, Igor, Vlad, Julia, Xenia, and everyone else I have met before and during the PhD. I am lucky to have you by my side regardless of the random places on Earth we end up in. Moreover, thanks to my AU colleagues who made this office so pleasant to work in.

Finally, I want to express gratitude to all the people of Ukraine. The only reason I have been able to do research in a seemingly normal world is their heroic effort to fight for such a world.

*Egor Namakonov,*
*Aarhus, October 27, 2026.*

# Contents

**Abstract** i

**Resumé** iii

**Acknowledgments** v

**Contents** vii

**I Overview** 1

**1 Open problems of liveness verification** 3
1.1 Ensuring liveness of concurrent programs with formal methods . . . 3
1.2 Existing approaches to verifying liveness . . . 7
1.3 Scope of this thesis . . . 10

**2 Background** 13
2.1 Iris . . . 13
2.2 Trillium . . . 19
2.3 Obligations-based reasoning . . . 22

**3 Results** 27
3.1 Structure of the thesis . . . 27
3.2 Personal contributions . . . 29
3.3 Rocq mechanization . . . 29
3.4 Future work . . . 29

**II Publications** 33

**4 Lawyer** 35
4.1 Introduction . . . 36
4.2 Obligations Model . . . 39
4.3 Program Logic . . . 51
4.4 Case Study: Modular Verification of a Fair Lock Client . . . 57

4.5 Related Work . . . . . 63
4.6 Conclusion, Discussion and Future Work . . . . . 65
4.7 Appendix . . . . . 66

**5 Verifying Wait-Freedom for Concurrent Higher-Order Programs 77**
5.1 Introduction . . . . . 78
5.2 Overview . . . . . 82
5.3 Background: Iris Logic . . . . . 85
5.4 Verifying Wait-Freedom in Lawyer Logic . . . . . 89
5.5 Case Studies . . . . . 92
5.6 Proving the Wait-Freedom Adequacy Theorem . . . . . 97
5.7 Discussion and Limitations . . . . . 103
5.8 Related Work . . . . . 105
5.9 Conclusion and Future Work . . . . . 108
5.10 Appendix . . . . . 108

**Bibliography 111**

# Part I

# Overview

# Chapter 1

# Open problems of liveness verification

In this chapter, I explain the research problem addressed by this thesis: formal verification of liveness properties. I start in Section 1.1 by explaining what liveness properties are, why formal methods are necessary to establish them, and what the challenges are in doing so. Then, in Section 1.2 I explain how the existing work addresses these challenges and what remains to be solved. Finally, in Section 1.3 I state which open challenges this work addresses.

## 1.1 Ensuring liveness of concurrent programs with formal methods

Concurrent programs—*i.e.* those consisting of several computations potentially executing in parallel and accessing shared resources—are prevalent in modern software. In some areas, such as operating systems, databases and distributed computing, concurrency is a part of system behavior, whereas in others, such as high-performance computing, it enables higher performance via parallelization.

However, it is very challenging to ensure that such programs are correct. There are various ways a concurrent execution can fail: having multiple program threads simultaneously access memory and corrupt it as a result [78], producing unexpected behaviors of a library [3], or failing to terminate when it is expected [74]. Some of these errors, such as one described in [78], may lead to failure in a life-critical system. It is thus crucial to make sure that concurrent programs behave correctly.

Strictly speaking, to show a program correct, one has to ensure the absence of errors in *all* possible executions of that program. Achieving this with *testing* (running the concrete program with specific inputs and comparing its output with the expected one) alone is rarely viable for realistic programs: the number of program executions to check can be huge or even infinite. This is especially relevant for concurrent programs, where thread interaction exponentially increases the number of possible executions. Therefore, an approach complementary to testing is *formal verification*.

In this approach, a program is represented as a mathematical object, which is then formally shown to exhibit the needed property, such as complete absence of errors. If the mathematical abstraction represents the concrete program precisely enough, it allows one to conclude the correctness of that program.

Formal methods are particularly useful for establishing *liveness* properties. These are the properties stating that "good things eventually happen during the execution", or more formally, the properties that can only be violated by full, possibly infinite execution traces. This is in contrast to *safety* properties stating that "bad things never happen during the execution" that can be violated by finite prefixes of full execution traces. Examples of liveness properties include termination of every program thread, fairness (*i.e.* that every non-terminated thread always takes a step eventually), non-blocking progress properties (guaranteeing that a given thread cannot be indefinitely blocked by others) such as wait-freedom, livelock-freedom (*i.e.* the absence of infinite busy-waiting) *etc.*, whereas safety properties include memory safety (that is, the memory is accessed according to the language-level abstractions), functional correctness (*i.e.* the program computes the expected result), deadlock-freedom[1]and non-interference (that is, confidential information is not indirectly revealed during the computation), among others. Importantly, the testing-based approaches are suitable for checking safety, as they assume the program under consideration terminates, but not for liveness, where termination (or more generally, eventually reaching a specific program state) itself is a property to be checked. Therefore, ensuring termination requires formally modeling the program and analyzing its potentially infinite executions.

Formally establishing liveness of concurrent programs has a number of challenges, the main one of which is *modularity*. By modularity of a verification approach I mean that verifying a composite program amounts to, up to a moderate extra effort, independently verifying program modules it consists of. Modularity enables the reuse of correct program modules and simplifies verification of large, realistic programs. However, the non-deterministic thread interaction makes it difficult to even state, let alone prove, properties of concurrent program modules. When it comes to liveness, this is further complicated by the intricate dependencies of modules on the execution environment and on each other, as we will see below.

The example in Figure 1.1 illustrates that liveness properties of an individual program module can be quite non-trivial. There, one would like to establish termination of `client` in Figure 1.1a—a program where one thread writes to a shared variable `flag` and another waits for this write to happen. To prevent unexpected behaviors arising from simultaneous access to `flag`[2], we make this access exclusive by protecting it with a lock. The concrete lock implementation is a parameter to `client`; as example implementations, we consider ticketlock and spinlock in Figures 1.1b and 1.1c. Note that termination of such a program requires a *fair scheduler*, *i.e.* the

[1] If the language includes blocking primitives.

[2] Such a situation is known as a *data race*. The exact definition and effect of a data race varies between the languages. In particular, in C it leads to undefined behavior of the entire program [7], whereas in Java it allows non-sequentially-consistent behaviors [1].

```
(* Client program is parameterized
   with newLock which must return
   an (acquire, release) pair *)

let left flag (acquire, release) =
  acquire (); flag := true; release ()

let rec right flag (acquire, release) =
  acquire ();
  let c = !flag in
  release ();
  if c then () else
  right flag (acquire, release)

let client () =
  let flag = ref false in
  let lk = newLock () in
  left flag lk || right flag lk
```

(a) Client using the lock

```
let newLock () =
  let (ow, tk) = (ref 0, ref 0) in
  let acquire () =
    let t = FAA(tk, 1) in
    let rec wait () =
      let o = !ow in
      if (t == o)
      then ()
      else wait ()
    in wait ()
  in let release () =
    ow := !ow + 1
  in (acquire, release)
```

(b) Ticket lock implementation

```
let newLock () =
  let l = ref false in
  let rec acquire () =
    if CAS(l, false, true)
    then ()
    else acquire ()
  in let release () =
    l := false
  in (acquire, release)
```

(c) Spinlock implementation

Figure 1.1: A program built using locks, written in an ML-like language. FAA and CAS are correspondingly "fetch-and-add" and "compare-and-swap" atomic primitives

guarantee that every thread that has not terminated yet is always eventually scheduled for execution. This is needed to ensure that *e.g.* the left thread eventually sets the flag.

To simplify the proof of `client` termination, we would like to reduce it to establishing termination of each function call in it. Since the lock implementation is a parameter to `client`, it cannot prove the termination of lock-related functions on its own—rather, the lock implementation must provide the termination guarantees of these functions. Among them, `acquire` has the most intricate guarantees. Let us first consider the ticketlock implementation in Figure 1.1b. First, it requires that every single previous call to `acquire` has been followed by *exactly one* call to `release`. Second, similar to above, it requires the scheduler to be fair, in particular, that any thread holding the lock is always eventually scheduled. With these two assumptions, a call to the ticketlock implementation of `acquire` eventually terminates. When used in `client`, these assumptions are indeed satisfied, which allows one to show the termination of `client`. Note, however, that termination of other implementations of `acquire` may require more assumptions. For example, the spinlock implementation in Figure 1.1c additionally requires that `acquire` is only being called a finite, fixed number of times; otherwise, a given call to `acquire` might never succeed. This

additional requirement cannot be satisfied by `client`, as the number of calls to `acquire` depends on the scheduler. Thus, the general specification for `acquire` that `client` relies on can include the first two termination conditions, but not the additional one required by spinlock. Then, the specification must include the mutual exclusion guarantee to ensure that accessing `flag` does not introduce unwanted behaviors. Stating all of these guarantees in the same specification, let alone using them in the `client` termination proof, is highly non-trivial.

Besides modularity, there are other challenges in verifying liveness of concurrent programs.

- "Termination" almost always means *safe* termination, *i.e.* that all threads successfully complete their execution without *e.g.* getting stuck in the process. Thus, the verification framework should support reasoning about safety properties as well.

- Second, the formal semantics of a language considered by the verification framework must be detailed enough to capture realistic programs. For example, while many works on verifying concurrency consider the parallel composition as a language primitive, real-world languages operate at the level of the fork-join model, which brings its own challenges for liveness verification. In fact, the parallel composition operator `||` used in Figure 1.1 is also a derived construct in our language that must be given a specification and verified as well.

- Third, note that the program in Figure 1.1 is *higher-order*: the `newLock` returns a closure over the internal lock state. Verifying it requires reasoning about first-class functions.

- Some liveness guarantees, such as non-blocking progress properties, describe the execution of a given program module *in an arbitrary environment*. Thus, a framework supporting these properties must provide a way to reason about unverified, possibly unknown code.

- A framework supporting all the above is bound to be a complex system on its own. Using it for ensuring program correctness requires trusting the framework itself. Instead of simply assuming that the framework is sound, it is better to make it *foundational* in some meta-level logic. That is, both the language semantics and the target property are expressed in that logic, and the verification framework merely provides a convenient tool for establishing the property in that logic, which removes the need for trusting the framework itself. Proof assistants, such as Rocq [2] and Lean [4], provide such meta-level logics, along with the mechanization tools that allow one to automatically check the correctness of resulting proofs.

- Finally, to ease the verification process, it is desirable to automate it to a certain degree.

## 1.2 Existing approaches to verifying liveness

Below, I explain how the challenges stated in Section 1.1 are addressed by the existing work in the area. I first consider approaches to verifying termination, as it is the most developed line of work, and the issues related to it apply to other liveness properties. Then, I consider approaches to verifying progress of individual program modules.

### Proving termination

In the context of verification of concurrent programs, "termination" usually means safe termination (as described in Section 1.1) of each program thread, possibly assuming fair scheduling (*i.e.* that every thread always eventually takes a step until it terminates). Note that bounded termination (*i.e.* termination in a fixed number of steps known in advance) is a safety property: proving it only requires considering finite executions up to a certain length. In the review below, I concentrate on proving unbounded termination, where the number of steps a program takes is not known in advance.

Many approaches to proving termination rely on establishing, in one way or another, a well-founded measure on program execution state which is then shown to decrease during the execution. This idea, originally proposed by Floyd [31], has been later developed under the name of *ranking functions* [84]. This line of work focuses on automatic inference of ranking functions for closed, sequential, first-order programs represented as transition systems [16, 102, 121]. Some works extend it to concurrent programs [72, 83, 86]. Some of the solutions [18] achieve a degree of modularity by establishing ranking functions for each module being used. However, this approach does not consider blocking concurrency and modular specification of *e.g.* lock libraries. Most of the works in this area have limited support for safety verification and are not mechanized in proof assistants (see Ströder et al. [114] and Meng et al. [87] for examples of exceptions).

The approaches above either try to automatically infer the termination measure, or require the user to provide it. An alternative to that is to design a type system where typing derivations embed the construction of that measure, and thus well-typed terms are guaranteed to terminate. The benefit of this approach is modularity: typing a term reduces to verifying its subterms. Moreover, some of the works listed below, such as those by Shoshi et al. [111] and Voirol et al. [125], automate the termination proofs with (limited) type inference. There are two main branches of such type systems. First, some of them establish termination of sequential programs written in pure functional languages [123, 125, 126]. These systems do not address the question of verifying fine-grained blocking concurrency. Second, there is a considerable number of type systems for variations of pi-calculus that account for liveness properties, including termination [21, 67, 111, 126]. These systems inherently target concurrent, possibly blocking programs. However, they also have a number of limitations. First, pi-calculus terms have a high degree of abstraction and are hard to use for modeling fine-grained imperative programs. Second, there are no type systems expressive

enough for capturing the complex termination conditions such as those of the lock library in Figure 1.1.

Another line of work developed as a part of the CertiKOS project [39, 40] models program modules as concurrent game strategies [107]. In this approach, program execution is represented by a trace of abstract events. An event represents executing either a language primitive or an entire function call, which simplifies reasoning about concurrent interleavings. The semantics of a single program module is then defined as a *strategy*, *i.e.* a function that extends the current execution trace with the next event. This approach is very general and supports verification of a variety of program properties, including liveness: a case study on MCS lock verification [65] shows that this approach is capable of establishing modular termination specifications of programs written in a C-like language. Moreover, this line of work is mechanized in Rocq. However, these specifications are noticeably more complex than those established by the approaches below. Moreover, proving liveness specifications such as one in [65] requires direct reasoning about the scheduler. Finally, this line of work has not been yet applied to verifying higher-order programs and thus is not capable of verifying closures in Figure 1.1. In a similar line of work, Ioannidis et al. [51] denote their language in a variation of Interaction Trees [127] that supports liveness reasoning. This work so far has not been applied to verifying blocking concurrency.

Then, termination of a program can be shown by refinement. In general, refinement means that all observable program behaviors are also observable for another program that serves as a specification. In the context of proving termination, one establishes termination-sensitive refinement, *i.e.* prohibiting non-terminating executions if they are prohibited by the specification program. Most of the solutions [33, 117] that establish termination-sensitive refinement are developed for verifying compilation correctness. Therefore, these solutions do not consider giving precise specifications for *e.g.* lock libraries. A notable exception is the work by Lee et al. [75] that refines a ticketlock implementation from Figure 1.1b to an abstract "lock specification" program. However, such a specification program is almost as complicated as the actual program that refines it and contains special primitives that expose the effects of the scheduler at the language level. In fact, the follow-up work by Lee et al. [76] simplifies working with such refinement by employing a more advanced program logic—an approach discussed in more detail below. All of the aforementioned works establishing refinement are mechanized in Rocq.

Finally, there are program logics for proving termination. Among them, there are logics for verifying sequential and concurrent programs with no thread interaction [8, 85, 112] and for verifying bounded termination [88]. However, these works do not cover the most challenging aspects of termination verification—thread interaction and scheduler fairness. Thus, another line of work [15, 52] considers languages with blocking primitives and ensures termination by enforcing a well-founded order on using these primitives. While these logics support verifying programs like `client` in Figure 1.1 by treating `acquire` and `release` as primitives, they are not sufficient for verifying the actual implementations of these lock functions. Therefore, the most advanced branch of program logics for termination addresses fine-grained blocking

concurrency. Among them, Fairis logic [118] supports a wide variety of liveness properties, including fair termination, but compromises on modularity. So far, all program logics [26, 28, 76, 104] capable of modular termination verification of fine-grained blocking concurrency are *obligations*-based—I cover this approach in more detail in Section 2.3. A considerable number of recent program logics for termination are mechanized in proof assistants. However, the modularity of these logics is limited compared to similar logics for safety verification in higher-order languages. In particular, none of the aforementioned works establishes a general lock specification similar to the one by Birkedal and Bizjak [12]. This lock specification quantifies over an arbitrary property protected by the lock (*i.e.* a property that is guaranteed to hold upon lock acquisition and is required to be reestablished before lock release), allowing the lock client to instantiate it as needed. The semantic mechanism that ensures soundness of such quantification is known [112] to be incompatible with liveness reasoning in general.

For completeness, I also briefly mention work on termination outside of the concurrent, possibly blocking setting. There has been work [20, 106] on using abstract interpretation for establishing termination of sequential programs. Also, the problem of proving termination has been explored in a probabilistic setting, leading to notions such as almost-sure termination [37].

### Proving progress of independent modules

Termination is a property of a closed program, not the individual modules it consists of. Despite most of the aforementioned approaches achieve modularity in a sense that every module is given a specification and verified independently, the end result of verification—proof of termination—is established only for composition of these modules. However, there are situations where the progress of an individual module is important on its own, regardless of the environment it executes in. For example, OS kernel threads must never become unresponsive, even if user-mode applications might do so. Therefore, it is important to verify progress of individual modules.

Liang and Feng [80] propose to characterize a module's progress based on how it is affected by the environment. When a module stalls because of its environment's actions, it is called *delay*, and when it happens because of the environment's inactivity, it is called *blocking*. That classification introduces four progress properties: *deadlock-freedom* and *starvation-freedom* both allow blocking, whereas *lock-freedom* and *wait-freedom* both prohibit it; only the first property in each pair allows delays. Lili, the framework introduced in the same work by Liang and Feng [80], is the only one so far that supports the verification of all four properties. It does so by refining the program module to an abstract concurrent object. However, this solution makes a simplification to the programming language that cannot be implemented in higher-order languages that we consider.

Verification of deadlock-freedom is a well-developed problem. It is often treated as a safety property: if the language under consideration has blocking primitives, one can track the threads that execute them and verify that the threads are never blocked all

together. There are a number of program logics and type systems (both for pi-calculus and realistic languages) that ensure deadlock-freedom in that way [42, 55, 56, 67, 97]. However, this approach does not help with verifying the absence of a *livelock*—a situation where threads technically continue execution, but in fact only by waiting for a certain event to happen. Approaches such as Lili and the one by Kobayashi [66] treat deadlock as a liveness property and thus allow one to rule out livelocks. Then, verification of starvation-freedom is a less developed problem; besides Lili, there is a bespoke rely-guarantee program logic [109] designed for that.

Among the properties of non-blocking concurrent programs, lock-freedom is studied the most. One approach to verifying it is by reducing lock-freedom to termination [50, 59, 101]. Besides that, lock-freedom has been verified with the use of temporal logic [120]. Wait-freedom is verified with a number of program logics [36, 99]; besides that, there exist universal constructions [19, 29] that convert a sequential implementation of a data structure into a wait-free one. Moreover, Herlihy et al. [44] suggested another non-blocking property—*obstruction-freedom*—that requires an operation to progress only in the absence of other concurrent operations. So far, only Li and Feng [79] have proposed a program logic for verifying obstruction-freedom.

In verifying the progress of a given module, all the aforementioned work makes a critical assumption that the environment of the module is *safe*, *i.e.* it does not modify the module's internal state in an unintended way. For example, Liang and Feng [80] restrict the language under consideration, whereas other works [36, 101] assume that all the interaction of the module with its environment satisfies a certain property. The former solution cannot be implemented for a realistic higher-order language; the latter means that the progress guarantee can only be established for a concrete, verified environment. Besides that, most of the existing work on progress verification considers first-order languages and is not mechanized (with some exceptions such as work on deadlock-freedom by Jacobs and Balzer [55]). Most of the aforementioned approaches provide some degree of modularity in proving the progress property in question: for example, wait-freedom of specifications by Pinto et al. [101] is preserved by sequential composition. However, since these works consider first-order languages, they do not provide means of proving that *e.g.* a mapping function on lists, specialized to a wait-free function argument, is also wait-free.

## 1.3 Scope of this thesis

In this thesis, I develop a solution for modular verification of liveness properties in higher-order languages. Specifically, I design a program logic inspired by the variation of the obligations-based approach implemented by Sassy [28]. To work around the modularity limitation mentioned in Section 1.2, I build on top of Trillium [118] that suggests a way to establish liveness properties when reasoning in a higher-order logic.

I use this logic for proving two kinds of liveness properties. First, I use it for establishing termination (bounded and unbounded, as well as under fair and unfair schedulers) of closed programs. Second, I show that a particular specification format

in this logic implies wait-freedom of a given operation, even if it executes in an unknown, higher-order context.

By building on top of Trillium (and the underlying Iris [62] framework) and by employing obligations-based reasoning, this work addresses almost all challenges listed in Section 1.1. The modularity of individual modules' liveness specifications is similar to that of state-of-the-art safety specifications. In particular, I verify termination of `client` in Figure 1.1 by relying on a general, liveness-aware lock specification similar to the one by Birkedal and Bizjak [12]; I prove this specification separately for the ticketlock implementation. For wait-freedom, modularity is achieved in verifying higher-order functions, such as a list mapping function applied to a wait-free function argument. Then, this program logic establishes safe progress (when applicable) and operates with a higher-order language with fork-based concurrency. It allows establishing wait-freedom of an operation in an unknown, higher-order context. All the results in the thesis are mechanized in the Rocq proof assistant [43] using the Iris Proof Mode [71]. The only challenge this work does not address is automation: this aspect of verification is less developed within program logics and only recently started to get more attention [90, 108]. Thus, the automation in this work consists of Rocq tactics only, which is on par with most of the existing works on Iris.

# Chapter 2

# Background

In this chapter, I explain the foundations that this work builds upon. The key component is the Iris separation logic framework discussed in Section 2.1. Then, in Section 2.2 I explain why the basic approach of Iris falls short when dealing with liveness, and introduce Trillium—the recent extension to Iris that allows, in theory, establishing liveness. To make it practical, one needs to instantiate it with a concrete formal system for liveness reasoning. For that, I choose the obligations-based approach described in Section 2.3.

In this thesis, I consider programs written in a higher-order ML-like language with general references and concurrency. Its semantics is provided further in the thesis; for now, it is sufficient to know that the expression reduction relation $\rightsquigarrow$ of this language is defined on pairs of an expression and a *heap* (partial function from locations to language values). For example, the reduction

$$(\ell := v,\ h[\ell \mapsto _]) \rightsquigarrow ((),\ h[\ell \mapsto v])$$

states that, for any heap $h$ with location $\ell$ present, and value $v$, the expression $\ell := v$ reduces to the unit value $()$ in a single execution step, while updating the value under $\ell$ to $v$. Then, these expression reductions are lifted to language *configurations* consisting of a *thread pool* (a list of expressions; the next reduction of each is determined by the evaluation context) and a heap, producing the small-step semantics $\rightarrow$.

## 2.1 Iris

In this section, I introduce Iris as a *concurrent separation logic for safety reasoning*. Technically, Iris only provides a minimal base logic that does not include any primitives specific for program verification—the user must derive them instead. This allows to use Iris for proving a variety of properties for programs written in various languages. However, a common instantiation of Iris, presented *e.g.* in Iris Lecture Notes [12], targets the aforementioned ML-like language for which it allows to prove functional correctness and safety (explained below). The work presented in this thesis uses a similar instantiation which is partially shown in Figure 2.1 and will be explained

$$
\begin{aligned}
P, Q ::= {} & \mathsf{True} \mid \mathsf{False} \mid P \wedge Q \mid P \Rightarrow Q \mid \forall x.\, P \mid \exists x.\, P \mid x = y && \text{(higher-order logic)} \\
& P * Q \mid \mathtt{o} \mid \mathtt{1} \mid && \text{(separation logic with ghost resources)} \\
& \{P\}\ (e : Expr)\ \{v.Q\}_{\mathscr{E}} \mid (\ell : Loc) \mapsto (v : Val) \mid \boxed{P}^{\mathscr{N}} && \text{(program logic)} \\
& \triangleright P \mid && \text{(step-indexing)} \\
& \ldots
\end{aligned}
$$

Figure 2.1: Selected propositions of Iris logic. *Expr*, *Val* and *Loc* are the sets of our language's expressions, values and location correspondingly.

throughout this section. Thus, here by "Iris" I will mean this specific instantiation and will use it to illustrate the main concepts of concurrent separation logic.

Broadly speaking, Iris is a example of *Hoare logic*—a formal system for reasoning about program behavior. The central notion of such systems, originally proposed by Hoare [48], are the propositions (later dubbed as *Hoare triples*) of the form $\{P\}\ e\ \{v.\, Q\}$. Intuitively, such a triple states that for any execution of program $e$ that starts when the precondition $P$ holds and terminates by reducing to some value $v$, the postcondition $Q(v)$ holds. The program logic provides Hoare triples for language's primitives and rules to compose them. By using these, in the end the user obtains a Hoare triple for the closed program. The following *adequacy theorem* states that proving such a Hoare triple within Iris implies safety and functional correctness of the closed program.

**Theorem 1** (Adequacy of Iris for safety, simplified)**.** *Let $e$ be a language expression and $\phi$ be a meta-level predicate on language values (such as being equal to some fixed value). If $\vdash \{\mathsf{True}\}\ e\ \{v.\, \phi\}$ holds, i.e. this triple is provable in Iris logic, then any execution of $e$ (with any starting heap) is safe (i.e. it executes safely without getting stuck), and* if *it terminates by reducing to some value $v$,* then *postcondition $\phi(v)$ holds.*

Establishing Hoare triples for simple programs amounts to composing triples for language primitives. These primitive triples denote how the program configuration changes on every execution step, thus internalizing the expression reduction relation $\rightsquigarrow$ in the logic. To account for heap, we use *points-to* proposition $\ell \mapsto v$ which states that the location $\ell$ currently stores the value $v$ and also assumes exlcusive ownership of that location (more on ownership below). Figure 2.3 shows a number of rules provided by Iris for primitive Hoare triples and for composing them. Using these rules, we can prove a Hoare triple for a simple sequential program, namely

$\{\mathsf{True}\}$ `let l = ref 0 in FAA(l, 1); !l` $\{v.\, v = 1\}$

```
{True}
let l =
  {True}
  ref 0
  {ℓ. ℓ ↦ 0}
in
{l ↦ 0}
FAA(l, 1);
{l ↦ 1}
!l
{v. v = 1}
```

Figure 2.2: Verification of a sequential program

IRIS-HT-REF
$\{\mathsf{True}\}\ \mathtt{ref}\ v\ \{\ell.\,\ell \mapsto v\}$

IRIS-HT-LOAD
$\{\ell \mapsto v_\ell\}\ !\ell\ \{v.\,v = v_\ell \wedge \ell \mapsto v_\ell\}$

IRIS-HT-SEQ
$$\frac{\{P\}\ e_1\ \{Q\} \qquad \{Q\}\ e_2\ \{R\}}{\{P\}\ e_1; e_2\ \{R\}}$$

IRIS-HT-FAA
$\{\ell \mapsto n\}\ \mathtt{FAA}(\ell,k)\ \{m.\,m = n \wedge \ell \mapsto (n+k)\}$

IRIS-HT-LET
$$\frac{\{P\}\ e_1\ \{v.Q\} \qquad \forall v.\ \{Q[v/x]\}\ e_2[v/x]\ \{R\}}{\{P\}\ \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2\ \{R\}}$$

IRIS-HT-CSQ
$$\frac{\{P'\}\ e\ \{Q'\} \quad P \Rightarrow P' \quad \forall v.\ Q'(v) \Rightarrow Q(v)}{\{P\}\ e\ \{Q\}}$$

Figure 2.3: Selected rules for basic Hoare triples

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

IRIS-HT-PAR
$$\frac{\{P_1\}\ e_1\ \{Q_1\} \qquad \{P_2\}\ e_2\ \{Q_2\}}{\{P_1 * P_2\}\ e_1 || e_2\ \{Q_1 * Q_2\}}$$

SEP-WEAKEN
$P * Q \Rightarrow P$

IRIS-HT-INV
$$\frac{\{R * \triangleright P\}\ e\ \{Q * \triangleright P\}_{\mathscr{E}\setminus\mathscr{N}} \qquad e \text{ is atomic} \qquad \mathscr{N} \subseteq \mathscr{E}}{\left\{R * \boxed{P}^{\mathscr{N}}\right\}\ e\ \{Q\}_{\mathscr{E}}}$$

IRIS-INV-ALLOC
$$\frac{\{P\}\ e\ \{Q * R\}}{\{P\}\ e\ \left\{Q * \boxed{R}^{\mathscr{N}}\right\}}$$

Figure 2.4: Concurrent separation logic of Iris

Figure 2.2 illustrates this proof by showing the intermediate assertions that hold between the program steps.

However, the rules in Figure 2.3 are not sufficient on their own. For example, they do not suffice to verify a sequence of FAA's accessing different locations. Indeed, it is not clear how to apply IRIS-HT-FAA (that assumes a single points-to as the sole precondition) when the heap contains multiple locations (which would be represented with multiple points-to in the logic). More generally, we need a way for composing the specifications for programs that operate on disjoint parts of the heap.

This specification style and reasoning rules are achieved with *separation logics*. In such logics, propositions denote *ownership of resources*, such as points-to. In general, resources are disjoint: the *separating conjunction* $P * Q$ denotes a resource that can be separated into two resoures satisfying $P$ and $Q$ correspondingly; for example, ownership of $x \mapsto 0 * y \mapsto 1$ comprises of ownership of $x \mapsto 0$ and $y \mapsto 1$ where $x$ and $y$ are guaranteed to be distinct memory locations. In contrast, $P \wedge Q$ denotes a single resource that satisfies both $P$ and $Q$. In general, resources are non-duplicable: $P \not\vdash P * P$ for an arbitrary $P$; for example, there can be only one points-to resource for any given location.

Separation logic is very useful for verifying concurrent programs. Namely, it enables to split the ownership of heap between the threads, verify them separately and then conclude the fact about the whole heap again. In particular, Figure 2.5 illustrates the proof of a Hoare triple for a program that features two threads incrementing their own locations. To verify it, we use the IRIS-HT-PAR rule that allows to distribute the resources between the two threads and then combine them back. The separation logic is also useful in the sequential setting: to verify *e.g.* the last load of $y$, we use IRIS-HT-FRAME to strip irrelevant resources (*i.e.* points-to for $x$) in order to apply the load rule. Note that the Iris logic is *affine*: the SEP-WEAKEN rule allows to discard resources, which is formally needed to establish the final postcondition in Figure 2.5.

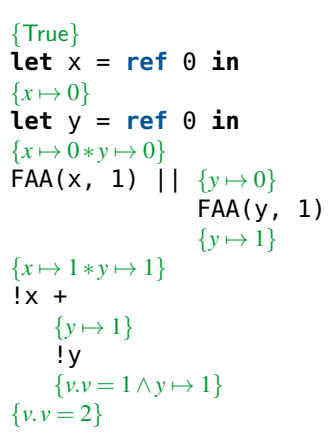



Figure 2.5: Verification of a program with disjoint parallelism

However, these rules are not sufficient to verify programs with shared state concurrency. Consider, for example, proving the same specification as in Figure 2.5 but for a program where threads operate on the *same* location:

$$\{\mathsf{True}\}\ \texttt{let l = ref 0 in (FAA(l, 1) || FAA(l, 1)); !l}\ \{v.\, v = 2\}$$

Intuitively, it must hold, as the location gets incremented twice. However, we cannot use IRIS-HT-PAR, as there is only one points-to resource, and both threads must use it.

In fact, we can verify a triple very similar to one above if, instead of reasoning about the heap directly with the points-to resource, we reasoned more abstractly. Namely, it is better to consider the *contributions* that each of the two threads make towards the total value at $l$. Consider the proof outline in Figure 2.6; for now, ignore the precondition and the initialization of location `l`. Once `l` is initialized to 0, contributions of both threads are 0 as well—expressed as ownership of two $\mathbf{o}$ resources. When a thread executes FAA, its own contribution—and the value under `l`—increases by 1; in the logic, this is expressed by replacing the $\mathbf{o}$ resource with $\mathbf{1}$. After both increments complete, we own two $\mathbf{1}$ resources, representing the total contribution of 2 to the `l` value; we expect that at that moment reading `l` will indeed return 2. Importantly, despite `l` is shared, now we can use IRIS-HT-PAR—the contribution resources can be distributed between the two threads and collected back. This is an example of *fictional separation* [24]—the contributions of each thread are treated as disjoint resources, even despite they both refer to the same location.

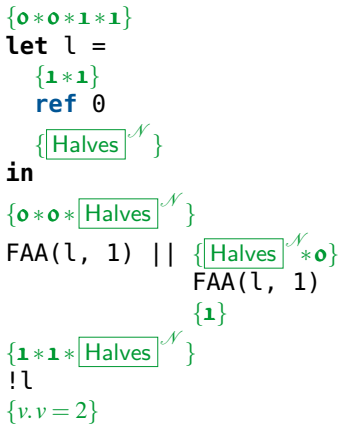



Figure 2.6: Verification of shared concurrency with the notion of fictional separation

Since Dinsdale-Young et al. [24], this concept has been implemented in a variety of separation logics [23, 25, 58, 62, 100].

To support the verification process as outlined in Figure 2.6, Iris provides the two key concepts: *ghost resources* and *invariants*. Ghost (*i.e.*, not directly tied to the program configuration) resources are chosen by the user. To conduct the proof outlined in Figure 2.6, we use the ghost resources $\mathtt{o}$ and $\mathtt{i}$. Compared to *e.g.* the points-to resources the composition of which is only valid for disjoint locations, the composition rules for ghost resources can vary. For example, the key rule for $\mathtt{o}$ and $\mathtt{i}$ is that they denote a contribution of *one of exactly two threads*, meaning that holding *more than two* instances of either resource is impossible:

CONTRIB-ZERO-HALF
$\mathtt{o} * \mathtt{o} * \mathtt{o} \Rightarrow \mathsf{False}$

CONTRIB-ONE-HALF
$\mathtt{i} * \mathtt{i} * \mathtt{i} \Rightarrow \mathsf{False}$

Then, an *invariant* $\boxed{P}^{\mathcal{N}}$ states that proposition $P$ holds on every step of execution. That is, each thread can assume that $P$ holds before taking a step, but must also prove that $P$ still holds after that step. Thus, resources that $P$ consists of are not owned by any thread—rather, they can be thought as "owned" by an invariant and borrowed by a thread for a single execution step. The *namespace* $\mathcal{N}$ is an invariant identifier explained below; in the thesis, it is omitted if irrelevant. A common use case for invariants is enforcing the aforementioned fictional separation by *connecting the state of a shared physical resource to ownership of ghost resources*. In our example, we would like to match each increment to $\mathtt{l}$ with an instance of $\mathtt{o}$ or $\mathtt{i}$ depending on whether or not the increment has happened. For this, we use the invariant $\boxed{\mathsf{Halves}}$ that maintains the following proposition:

$$\mathsf{Halves} \triangleq \exists n.\, l \mapsto n * (n = 0 * \mathtt{i} * \mathtt{i} \lor n = 1 * \mathtt{o} * \mathtt{i} \lor n = 2 * \mathtt{o} * \mathtt{o})$$

Before the increments happen, two $\mathtt{o}$ resources should be held by threads. Therefore, we establish the invariant by providing *the rest* of resources of the precondition in Figure 2.6:

$$\{\mathtt{i} * \mathtt{i}\}\ \texttt{ref}\ \mathtt{0}\ \left\{l.\boxed{\mathsf{Halves}}^{\mathcal{N}}\right\}$$

The main rule used to prove this triple is IRIS-INV-ALLOC that allows to move the resources held by a thread to an invariant. In our case, we move the two $\mathtt{i}$ resources and a newly created $l \mapsto 0$ resource into an invariant. We identify this invariant with an arbitrary namespace $\mathcal{N}$.

To verify FAA's made by each thread, we use IRIS-HT-PAR. Crucially, invariant propositions can be soundly duplicated: $\boxed{P}^{\mathcal{N}} \vdash \boxed{P}^{\mathcal{N}} * \boxed{P}^{\mathcal{N}}$; each of the concurrent threads can thus obtain a copy of the invariant. Thus, in each thread we prove

$$\left\{\mathtt{o} * \boxed{\mathsf{Halves}}^{\mathcal{N}}\right\}\ \texttt{FAA(l, 1)}\ \{\mathtt{i}\}$$

To update $\mathtt{l}$, we now need to use, or *open*, the invariant with the rule IRIS-HT-INV, which has a number of side conditions. First, an invariant can only be opened for a

single atomic step, such as FAA. Then, an invariant can only be opened once per step; otherwise, IRIS-HT-INV could be used to duplicate non-duplicable resources and thus become unsound. This is enforced by the *mask* $\mathcal{E}$ that annotates the Hoare triple and consists of the namespaces of invariants that can be opened in proving the triple—this is denoted by the $\mathcal{N} \subseteq \mathcal{E}$ condition. The full mask, *i.e.*, one that allows to open any invariant, is denoted as $\top$ or is simply omitted. Finally, the later modality $\triangleright$ will be explained in Section 2.2; for so-called "timeless" invariants[63] such as Halves (and all the invariants formally presented in this thesis[1]) it can be ignored. Thus, after applying IRIS-HT-INV, we need to prove

$$\{\mathtt{o} * \mathsf{Halves}\}\ \mathsf{FAA(l,\ 1)}\ \{\mathtt{1} * \mathsf{Halves}\}_{\top \setminus \mathcal{N}}$$

In general, when accessing the invariant, we do the following: conclude the current state of the shared physical resource using the opened invariant and other resources, then take the execution step to update the physical resouce, and finally match the updated resource with the available ghost resources to reestablish the invariant. By combining $\mathtt{o}$ and Halves, we can conclude (using CONTRIB-ZERO-HALF) that the current value under l is 0 or 1. Assume it is 0 (the other case proceeds similarly); then proving the above triple can be reduced to the following:

$$\{\mathtt{o} * l \mapsto 0 * \mathtt{1} * \mathtt{1}\}\ \mathsf{FAA(l,\ 1)}\ \{\mathtt{1} * l \mapsto 1 * \mathtt{o} * \mathtt{1}\}_{\top \setminus \mathcal{N}}$$

Here, we essentially swap $\mathtt{o}$ (held by the thread) and $\mathtt{1}$ (stored in the invariant) to reestablish Halves. We prove this triple using IRIS-HT-FRAME and IRIS-HT-FAA.

Finally, we verify the final read of l by proving

$$\{\mathtt{1} * \mathtt{1} * \boxed{\mathsf{Halves}}\}\ \mathtt{!l}\ \{v.v = 2\}$$

Similarly to above, we open the invariant and conclude that l indeed points to 2 using CONTRIB-ONE-HALF.

Note that the notion of ghost resources is much more powerful than presented here. First, it is not necessary to pre-allocate these resources and provide them in the precondition—they can be allocated dynamically; thus, one can prove the $\{\mathsf{True}\}\ \ldots\ \{v.v = 2\}$ triple for the program in Figure 2.6. Second, ghost resources provide rules for sound *updates*; with it, Figure 2.6 can be verified with a single contribution resource $\mathbf{contrib}(n : \mathbb{N})$ that generalizes $\mathtt{o}$ and $\mathtt{1}$. I omit these aspects of Iris resources, because the majority of ghost resources used in this thesis come from the Trillum framework and have their own, custom laws.

Finally, the entire Iris framework—language semantics, base logic and the instantiation shown in Figure 2.1, proof rules and adequacy – is mechanized in Rocq proof assistant [43] with Iris Proof Mode [71]. Here, the meta-level propositions about program executions are expressed in Calculus of Inductive Constructions [98]. To establish them, the user proves Iris specifications with the custom proof mode that embeds the separation logic reasoning into the general proof assistant interface.

[1] Except for the location case of value relation in Definition 15.

## 2.2 Trillium

In this section, I explain why Iris logic is useful for proving safety, but not liveness properties. In particular, I explain, on a high level, the model of Iris propositions that ensures their soundness, and why this model is geared towards representing safety properties. Then, I introduce Trillium—a recent extension of Iris that addresses this issue—and illustrate proving program liveness with it. Finally, I explain why the Trillium methodology on its own does not solve the challenge of modular liveness reasoning.

First of all, the flexibility of Iris logic requires some caution. Note that the propositions such as Hoare triples and invariants are *impredicative*: they are parameterized by arbitrary propositions of Iris logic, potentially including other Hoare triples and invariants. Moreover, they can reference themselves: Svendsen and Birkedal [115] show an example of a module specification that relies on a lock specification—which in turn controls the access to a resource that mentions the module specification again. However, allowing such self-referential specifications with no restrictions leads to liar-like paradoxes and makes the logic unsound [63]. Thus, there should be a mechanism that ensures the soundness of impredicative definitions.

Iris achieves soundness of impredicative propositions by employing a step-indexed model [10, 63]. Intuitively, in such a model validity of propositions is established *for a fixed number of remaining execution steps*. With this intuition, a proposition $P$ can be defined recursively by requiring that it continues to hold *after one step of execution*, expressed with *later modality* $\triangleright P$. If the recursive occurences only occur under later, Iris guarantees the soundness of such a definition. This modality appears when opening invariants (see IRIS-HT-INV), but often can be eliminated [63]; moreover, recent Iris developments [113] provide a more flexible way of working with it.

Now, let's take a step back from the Iris model and see how the approaches listed in Section 1.2 ensure termination of a program. Among them, a common idea is to take some process (which we will call a *model*) known to be terminating and *refine* a program execution to that process, *i.e.* match every program step to a step in that process. Indeed, in the approaches based on program refinement, the model is another program, whereas in approaches based on ranking functions the model is an abstract well-founded sequence of *e.g.* ordinals.

Such a refinement can be defined in (step-indexed) program logic. Namely, we can define some refinement predicate $\mathsf{Refines}$ that establishes correspondence between the current program configutation $c$ and the state $\delta$ of the model, and moreover requires to preserve this correspondence on every execution step by taking a step in the model:

$$\mathsf{Refines}(c,\delta) \approx \ldots (* \text{ some resource representing that } c \text{ corresponds to } \delta *) \ *$$
$$\forall c'.\, c \to c' \Rightarrow \exists \delta'.\, \delta \to \delta' * \triangleright \mathsf{Refines}(c',\delta') \ (* \text{ correspondence is preserved} *)$$

Indeed, the works that establish program refinement in Iris, such as Frumin et al. [32], use a similar predicate that internalizes refinement. The later modality enables

the use of advanced Iris features such as impredicative invariants while proving the refinement.

By proving $\mathsf{Refines}(c_0, \delta_0)$ for the initial configuration $c_0$ and some initial model state $\delta_0$, one would expect to conclude that every execution starting from $c_0$ is matched, on every step, by some model trace starting from $\delta_0$. However, proving $\mathsf{Refines}(c_0, \delta_0)$ in a step-indexed logic only establishes a weaker property—a *finite approximation* of the refinement, *i.e.* refinement for any finite prefix of the execution. This is not sufficient for proving termination: finite approximation does not say anything about infinite executions the absence of which we try to prove. Indeed, in step-indexed logics it is possible to refine a program that always runs infinitely long to an always terminating model [112, 118].

There are a couple ways to solve this problem. One option [112] is to change the definition of the step-index in a way that allows proving the needed whole-execution refinement. This approach, however, has not been applied to verifying termination of concurrent, blocking programs.

The approach I take in this thesis is based on the observation that *finite approximation implies the whole-execution refinement for a certain class of refinements*. The Trillium framework [118] is based on this observation. This framework is parameterized by a user-chosen model (an arbitrary labeled transition system) and the refinement relation that satisifes the so-called "relative image finiteness" restriction. By picking an appropriate model and a refinement relation, one can verify a variety of execution trace properties, including termination, which is formalized by the following adequacy theorem (simplified; see Timany et al. [118] for exact definitions of refinement and relative image finiteness).

**Theorem 2** (Adequacy of Trillium, simplified)**.** *Let $\mathscr{M}$ be a labeled transition system and $\xi$ be a relative image finite relation on pairs of finite execution and finite $\mathscr{M}$ traces. Let $e$ be a language expression and $\delta_0$ a state of $\mathscr{M}$. If $\vdash \left\{ \underset{\mathscr{M}}{\circ}\, \delta_0 \right\} e \left\{ \mathsf{True} \right\}^{\tau}$ holds, then any, possibly infinite, execution trace starting from the initial configuration $([e], \emptyset)$ refines ($\hat{\preccurlyeq}_{\xi}$ in terms of Timany et al. [118]) some trace of $\mathscr{M}$ starting from $\delta_0$.*

Above, $\underset{\mathscr{M}}{\circ}\, \delta$ is a ghost resource stating that the current state of the model $\mathscr{M}$ is $\delta$. With that, the Hoare triple from Theorem 2 can be seen similar to Refines predicate above; proving it requires matching every execution step with a transition in the model. Note that the Hoare triples of Trillium are parameterized with the thread identifier: when proving the refinement, it is important to track the thread identity.

As a simple example, consider using Trillium for verifying termination of the program `let l = ref 0 in FAA(l, 1); !l` the safety of which has been shown earlier in Section 2.1. For that, we instantiate Trillium with the model Decr: its states are natural numbers, labels are irrelevant (*e.g.* unit values), and the transition relation is "greater-than". Thus, its traces are decreasing (and thus finite) sequences of natural numbers. With any refinement relation, including the trivial always-true, any program that refines such model is guaranteed to terminate.

To verify the aforementioned program, we will prove the triple from Theorem 2. For that, we choose the initial model state to represent the total number of execution steps, *i.e.* $\delta_0 \triangleq 4$, and choose $\xi$ to be the always-true relation. Since proving such triple requires matching execution steps and model transitions, the proof rules of Trillium feature both the resources of Iris (discussed in Section 2.1) and model state resources; the concrete proof rules depend on exact Trillum instantiation. For our example with the Decr model, *e.g.* the rule IRIS-HT-LOAD is replaced by the following:

$$\text{TRILLIUM-DECR-HT-LOAD}$$
$$\left\{\ell \mapsto v_\ell * \underset{\mathsf{Decr}}{\circ}(n+1)\right\}\ !\ell\ \left\{v.v = v_\ell \wedge \ell \mapsto v_\ell * \underset{\mathsf{Decr}}{\circ} n\right\}^{\tau}$$

It states that a physical step must be justified with a transition in the Decr; since the transition relation of Decr is "greater-than", the current state must be a positive natural number to allow the transition. For other models, the concrete rules will be different, but in general they will involve some resources describing the current state of the model. With the rules similar to TRILLIUM-DECR-HT-LOAD, verification proceeds similarly to Figure 2.2.

The Trilllium methodology allows to verify liveness in higher-order concurrent and distributed languages [47, 118], but it has a number of issues. The first issue is that Trillium by itself does not help to choose an appropriate model and the refinement relation. Existing Trillium-based projects require the user to choose an ad-hoc model and relation relevant to the particular example. Yet, similar to how Iris logic applies to variety of safety verification projects, it is desired to choose a general model and corresponding program logic that applies to many examples. The other issue is the lack of modularity. Trillium requires to choose the "global" model for the entire closed program—but we want to verify each program module independently of others, and this requires refining module's execution to some unknown (from the module's perspective) global model. In a perfect scenario, we would like to verify each program module with its own model and then compose these models to obtain the global one. However, it is not clear what transition systems should represent "a model of a module", how to compose them and combine the program logics for each of them[2]. Moreover, the aforementioned relative image finiteness restriction requires workarounds in certain cases [118].

A concrete example that is hard to concisely verify with Trillium alone is the program from Figure 2.5. At the first glance, the approach with the Decr model described above should work, as the program seems to execute only finitely many primitives. However, this is not true: the program contains a parallel composition operator $||$. This is not a language primitive—rather, it is implemented using busy-waiting, the termination of which relies on fair scheduler. To account for that, one should define a model with the "fair termination" property and verify the program

[2]In fact, a significant part of working on this thesis has been dedicated to solving these exact questions—unfortunately, with no positive result. In my experience, attempting to verify each module with a completely arbitrary model requires a lot of trace-level reasoning and abandons many benefits of separation logic.

```
let flag1 = ref false in
flag1 := true || while (not !flag1) {}
```

(a) Using parallel composition

```
let flag1 = ref false in
let flag2 = ref false in
Fork(
  while (not !flag1) {};
  flag2 := true
);
flag1 := true;
while (not !flag2) {}
```

(b) With parallel composition unfolded

Figure 2.7: An example of program where progress of threads is mutually dependent

with it; indeed, this is how some of examples of Trillium are verified [118]. However, it complicates the model and requires to unfold the constructions such as `||`. Rather, one would like to keep these construction opaque and verify the programs using them with simple proof rules, keeping the model simple. In turn, soundness of these proof rules should be verified separately.

In a sense, Trillium is only as useful as the model it is instantiated with. This model must be expressive enough to capture a variety of programs, provide a convenient program logic, satisfy the technical restrictions of Trillium, and, in a way, allow to be composed from the smaller models. Below, I explain the inspiration for such a model in the context of proving termination.

## 2.3 Obligations-based reasoning

When it comes to modular liveness verification, the most challenging programs are blocking (see Section 1.2)—those the progress of which depends on the environment making progress as well. The reason for that is the absence of a clear decreasing measure that justifies the program making progress towards something: such a program, in isolation from its environment, can run forever. Modular specifications for such a program thus must contain conditions on its environment under which the program is guaranteed to progress.

As an example, consider the program in Figure 2.7a where one thread writes to a shared variable and the other thread, running in parallel, waits for this write to happen. In fact, in our language the parallel composition is a derived construct; by unfolding it, we obtain a program similar to one in Figure 2.7b. There, after writing to the flag, the main thread waits for the forked thread to finish its execution. This example can be seen as a simplified version of the lock client program from Figure 1.1: the waiting of the forked thread is similar to how the left thread from Figure 1.1 waits for the right thread to release the lock, and similarly for the waiting of the main thread.

It is easy to show the termination of the closed program in Figure 2.7b assuming fair scheduler. Indeed, once the main thread is scheduled, it will set `flag1`, which will terminate the loop in the forked thread. Then, by a similar argument, the `flag2` will

be set and the loop in the main thread will terminate. However, it is more desirable to verify the program in a modular way by considering the program components separately. In fact, there are multiple approaches for decomposing the program for that. One approach is to consider the two thread programs in Figure 2.7b after unfolding the parallel composition. Another, more useful approach, is to verify the parallel composition operator and then use it to verify the program in Figure 2.7a. Below, I follow the first, easier approach. Proving a general, expressive specification for the parallel composition is surprisingly tricky; I do it as a part of the technical development for the thesis.

To verify the termination of threads in Figure 2.7b, we need to specify what actions each of the threads progresses towards and which actions of other threads relies upon. For example, the forked thread will set `flag2` assuming some other thread sets `flag1`. Then, to conclude termination of these two threads running together, it is sufficient to check that these actions can be ordered in an acyclic fashion.

To support the above modular reasoning about conditional progress, modern separation logics [26, 28, 76, 104] rely on the notion of *obligations*. An obligation is an abstract representation of an action that some thread must perform in the future. The other threads can rely on that obligation being fulfilled (implicitly assuming fair scheduling) and take their own obligations that will be fulfilled afterwards. The key part of these logics is the obligations resource, often written as $\mathsf{obls}_\tau\,\mathscr{O}$, stating that the thread $\tau$ holds the set of obligations $\mathscr{O}$. Crucially, obligations can depend on each other: $ob_1 \prec ob_2$ means that fulfilling the obligation $ob_1$ is conditional on $ob_2$ being fulfilled before. The obligations order $\prec$ must be well-founded to rule out circular and infinite dependencies. Adequacy theorems of these logics reduce proving termination of a program (under a fair scheduler) to proving a Hoare triple with $\mathsf{obls}_\tau\,\emptyset$ in the postcondition; this ensures that whenever a thread terminates, it does not have pending obligations that other threads might rely on.

The notion of ordered thread obligations is very expressive and allows to verify various blocking programs in the same framework. In particular, despite the details of the aforementioned logics vary, all of them[3]allow to *derive* a number of rules that cover simple busy-waiting used in Figure 2.7. More specifically, in these logics a busy-waiting location can be associated with an obligation: writing to it fulfills the associated obligation, and until that happens, this obligation justifies the busy-waiting of other threads. Formally, these logics allow to derive a duplicable resource $\mathsf{BW}(\mathtt{l}, ob)$ (asserting that writing to the busy-waiting location `l` fulfills the obligation $ob$), along with the following rules (here I use the Trillium variant of Hoare triples indexed with a thread identifier):

[3]In case of Lilo [76], the obligations resource will be a wrapper over another resource, and the programs will be annotated with yields.

$\{\mathsf{obls}_\tau\{ob_1\} * \mathsf{BW}(\texttt{flag1}, ob_1) * \mathsf{BW}(\texttt{flag2}, ob_2)\}$
`  flag1 := true; while (not !flag2) {}` (spec-main)
$\{\mathsf{obls}_\tau\emptyset\}^\tau$

$\{\mathsf{obls}_\tau\{ob_2\} * ob_1 \prec ob_2 * \mathsf{BW}(\texttt{flag1}, ob_1) * \mathsf{BW}(\texttt{flag2}, ob_2)\}$
`  while (not !flag1) {}; flag2 := true` (spec-forked)
$\{\mathsf{obls}_\tau\emptyset\}^\tau$

Figure 2.8: Specifications for two threads in Figure 2.7b

BW-WAIT
$\{\mathsf{obls}_\tau\mathscr{O} * \mathsf{BW}(\mathtt{l}, ob) * \forall ob' \in \mathscr{O}.\, ob \prec ob'\}\ \mathtt{while}(\mathtt{not}\,\mathtt{!l})\{\}\ \{\mathsf{obls}_\tau\mathscr{O}\}^\tau$

BW-SET
$\{\mathsf{obls}_\tau\mathscr{O} * \mathsf{BW}(\mathtt{l}, ob) * ob \in \mathscr{O}\}\ \mathtt{l} := \mathtt{true}\ \{\mathsf{obls}_\tau\mathscr{O} \setminus \{ob\}\}^\tau$

With these rules, one can prove the specifications in Figure 2.8 for individual threads of the program in Figure 2.7b, expressing their termination being dependent on the environment. Note that in the specifications above the obligations $ob_1$ and $ob_2$ are (implicitly) universally quantified, like the thread identifier $\tau$. This makes the specifications modular: they can be used for any obligations the order on which satisfies the preconditions. To verify Figure 2.7b using the thread specifications above, we can indeed create two obligations $ob_1 \prec ob_2$ upon initalizing the flags `flag1` and `flag2`; initially both obligations are held by the main thread, *i.e.* we obtain $\mathsf{obls}_\tau\{ob_1, ob_2\}$ which is subsequently split into two obligations resources for two threads. Also at that moment, we obtain the $\mathsf{BW}(\texttt{flag1}, ob_1)$ and $\mathsf{BW}(\texttt{flag2}, ob_2)$ resources. I do not provide the formal rules used during the initialization: some of the aforementioned works [26, 76] do not have rules for forking and obligations transfer, which are needed to verify Figure 2.7b; these works are capable of verifying slight variations of this program.

As said above, the rules BW-WAIT and BW-SET for simple busy-waiting are derived; the obligations-based reasoning covers more complicated scenarios. For example, an obligation can be tied not to just writing `true`, but to an arbitrary condition. Moreover, a waiting thread can receive an arbitrary resource after waiting finishes. Then, more complicated examples such as Figure 1.1 require the obligations resource to change before, during, and after the waiting. Finally, verifying programs in languages with fork-based concurrency, such as ours, requires transfering obligations to forked threads. All of these scenarios are covered by (some of) the modern obligations-based logics.

The details of obligations reasoning and properties being established vary between concrete logics.

- TaDA Live [26] is designed to verify fair termination of blocking first-order programs. To support that, the Hoare triples in this logic explicitly state the *environment liveness assumptions* that describe what the environment must do in order for the given program to terminate. These liveness assumptions are stated in terms of state of shared resources: *e.g.*, a liveness assumption for busy-waiting states that the environment eventually writes `true` to the corresponding location. Specifically, the state of shared resources is controlled by *protocols* (an extension of those by Pinto et al. [100]) which dictate when and how a given resource can be updated. Thus, obligations annotate the individual updates of a protocol: an obligation owner is required to eventually update the shared resource as prescribed. This work features a variety of case studies on verifying fair termination modularly, including an example (lock-coupling set) not supported by the works listed below.

- Reinhard and Jacobs [104] continue the line of work [41, 105] on obligations-based logics that reduce proving termination to safety. For that, they instrument the programming language. First, the language configurations are extended to explicitly store thread obligations and *fuel* consumed during execution. Second, the language is extended with new primitives that operate with obligations and fuel. Importantly, since executing the program consumes fuel, the instrumented language does not have infinite fair safe executions. Obligations allow to replenish fuel to justify busy-waiting, but without breaking this guarantee. The instrumentation can be erased without affecting the program behavior. Together, it means that proving safety of an instrumented program implies fair termination of the original program. The novelty of Reinhard and Jacobs [104] specifically is supporting busy-waiting (as opposed to relying on blocking primitives), and also combining this approach with the previous work on verifying non-blocking concurrency [52]. Recently, this line of work has been further extended by Sassy [28] which has reached the TaDA Live-degree of modularity by parameterizing reusable program modules with ghost code to be executed before and upon linearization points. The aforementioned works have Verifast [54] implementations.

- The distinguishing feature of the Iris-based Lilo logic [76] is that it establishes not termination, but rather per-thread (assuming there are no forks) contextual refinement to a specification program. When every thread of the specification program is trivially terminating, Lilo essentially establishes (fair) termination. Besides that, Lilo also supports verification of some non-terminating programs. Another difference is that Lilo is designed to verify cooperative concurrency with explicit yields. At the low level, the logical primitives of Lilo are quite different compared to those of TaDA Live and Sassy line of work (*e.g.*, obligations resource explicitly mentions the proposition to be eventually established,

fuel is tied to the specific obligation, $ob_1 \prec ob_2$ is an Iris resource and not a meta-level proposition *etc.*). However, at the higher level the expressivity of Lilo *wrt.* termination reasoning is very close to that of other works.

The main issue with existing obligations-based logics is the limited expressivity compared to state-of-the-art logics for safety, as mentioned in Section 1.2. Most of them [26, 104] are first-order, which precludes *e.g.* using Hoare triples in invariants and other Hoare triples. Lilo stratifies propositions by levels, which increases expressivity, but is still too restrictive to define propositions that arbitrarily quantify over other propositions (such as the example by Svendsen and Birkedal [115] mentioned in Section 2.2). Sassy is designed to work with any sufficiently expressive safety logic such as Iris, but requires to extend the language under consideration with auxiliary primitives. Among these logics, only Lilo is mechanized in Rocq.

Finally, despite obligations-based reasoning is mostly used for verifying termination of blocking programs, they serve an important role in verifying non-blocking concurrency as well. First of all, obligations (as implemented in this thesis and *e.g.* by Sassy) are a conservative extension of fuel-based approach: when a thread does not hold the obligations resource, it can still burn fuel provided to it without waiting for other threads. With that, the specifications of blocking and non-blocking programs can be verified and composed in the same framework, with the only difference being whether the obligations resource is mentioned in the specification or not; this is discussed in Section 5.10. Second, while obligations are not used by the *user* that verifies non-blocking programs, they turn out to be very helpful for the *developer* of the program logic. Indeed, the adequacy theorem of our wait-freedom verification approach from Chapter 5 relies heavily on the manipulations with the obligations resource.

# Chapter 3

# Results

This thesis is based on and includes the following publications.

- Egor Namakonov, Justus Fasse, Bart Jacobs, Lars Birkedal, and Amin Timany. Lawyer: Modular Obligations-Based Liveness Reasoning in Higher-Order Impredicative Concurrent Separation Logic. *Proceedings of the ACM on Programming Languages*, 10(OOPSLA1):132:1099–132:1125, April 2026. doi: 10.1145/3798240 [94]

- Egor Namakonov, Lars Birkedal, and Amin Timany. Verifying Wait-Freedom for Concurrent Higher-Order Programs. *LIPIcs, Volume 372, ECOOP 2026*, 372:20:1–20:29, 2026. ISSN 1868-8969. doi: 10.4230/LIPICS.ECOOP.2026.20 [92]

## 3.1 Structure of the thesis

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

Chapter 4 contains the text (up to minor formatting changes) of Namakonov et al. [94] with technical appendices. This work provides the foundation for both the termination and wait-freedom reasoning.

An overwhelming problem of impredicative separation logics, such as Iris, is the lack of support for liveness reasoning. Existing solutions that aim to address it compromise either on expressivity, on the types of supported liveness properties, on mechanization in proof assistants, or on modularity. The latter is the approach of the Trillium framework [118] that refines program executions to traces of a labeled transition system and derives liveness properties from such refinement. This transition system must be chosen upfront for the closed program, thus making this approach non-modular.

Another existing solution that achieves modularity is Sassy [28]—an obligations-based logic that develops a modular specification format for concurrent blocking

libraries. Technically, this is achieved by instrumenting the language semantics in a way that guarantees fair termination. Sound reasoning about such instrumented programs is highly non-trivial, and thus so far Sassy has only been implemented in a non-foundational VeriFast tool [54].

Our work combines the two main ideas of Trillium and Sassy: refinement to a transition system and obligations-based reasoning. Namely, we develop Lawyer—a Trillium-based program logic for proving termination of fine-grained blocking and non-blocking concurrent programs. These proofs are done by refining the program executions to traces of Obligations Model—a transition system that models the obligations-based instrumentation of Sassy. The challenge of implementing the obligations-based reasoning this way is the technical limitation of Trillium that restricts the kind of transition systems used for refinement; we work around this limitation and make our logic expressive for almost all use cases.

As a result, Lawyer inherits the impredicative program logic of Iris and Trillium, while being expressive enough for supporting modular liveness Sassy-style specifications. We exemplify Lawyer by verifying, besides smaller case studies, the lock library and a client program that uses it, in a modular way. The logic of Lawyer, the underlying instantiation of Trillium and the case studies are mechanized in Rocq.

### Verifying Wait-Freedom for Concurrent Higher-Order Programs

Chapter 5 contains the text (up to minor formatting changes) of Namakonov et al. [92] with technical appendices. This work builds on top of Lawyer and repurposes its logic for verifying another kind of liveness properties.

Wait-freedom is the strongest progress guarantee for operations on concurrent data structures: it ensures that every call to the operation terminates, regardless of code executing concurrently and scheduling. It is not always guaranteed: often the implementations use locks to avoid unwanted concurrent interference, or retry the operation upon such interference. Wait-free implementations avoid both of these and are thus known for being notoriously difficult, and so it is desirable to verify their correctness formally.

Verification of wait-freedom has been addressed for first-order languages, but not yet in the more realistic higher-order setting, where an operation can be stored in the heap and passed around as a value by an arbitrary higher-order client program. Existing approaches that assume first-order languages do not scale to this setting.

In this work, we define and verify wait-freedom of higher-order concurrent programs by employing the Lawyer program logic. This involves a number of challenges. First, we formally define the notion of wait-freedom that accounts for the higher-order nature of the language. Second, Lawyer is designed for verifying closed programs, whereas we only verify the wait-free operation without the client that uses it. Thus, we prove a novel adequacy theorem that establishes wait-freedom for the verified operation only. In turn, this requires reasoning about arbitrary client programs, which we implement by using logical relations inspired by those used for proving robust safety.

We demonstrate our approach by verifying a number of case studies. Moreover, we discover that realistic algorithms exhibit slightly different progress guarantees, which is not addressed by existing work. We adapt our approach for supporting such algorithms and verify case studies for them as well.

## 3.2 Personal contributions

I am the main author of both Namakonov et al. [94] and Namakonov et al. [92]. In particular:

- For Namakonov et al. [94], I have implemented the Obligations Model and the Lawyer logic, verified the case studies and done the Rocq mechanization. Justus Fasse contributed to adapting the semantics and logically atomic specifications of Sassy in our setting, and to verifying the case studies. Bart Jacobs, Lars Birkedal and Amin Timany contributed to writing and supervised the project.
- For Namakonov et al. [92], I have defined wait-freedom at the meta-level, identified the specification format, proved the adequacy theorem, verified the case studies and done the Rocq mechanization. Lars Birkedal and Amin Timany contributed to writing and supervised the project.

## 3.3 Rocq mechanization

The Rocq mechanization is provided in the artifacts accompanying the papers: Namakonov et al. [95] and Namakonov et al. [93]; the latter has received the Distinguished Artifact Award of ECOOP'26. Moreover, the up-to-date mechanization of both Lawyer and the wait-freedom project is available on GitHub [5]. The mechanization uses a fork of Trillium which is also available on GitHub [6].

## 3.4 Future work

There are multiple directions for future work—the immediate improvements to Lawyer and wait-freedom verification, as well as bigger extensions for verifying more liveness properties, potentially in a different setting.

This section assumes familiarity with the content of the papers in Part II.

### Extending the termination verification

An important immediate improvement to termination verification with Lawyer is the support for arbitrary fuel exchanges. At the moment, they are not supported fully due to the relative image-finiteness restriction of Trillium, for which Lawyer provides the workaround of "exchange bound" (see the "less-or-equal" restriction of OMS-EXCHANGES-FUEL and Section 4.2 for more detailed explanation). Yet, this

workaround is not in general sufficient for verifying programs the execution time of which depends on runtime information, *e.g.*, traversing a linked list; these programs arise in the wait-freedom verification as well.

One way to support these programs is to employ a transfinite model and eliminate the relative image-finiteness restriction altogether. Another way is to exploit the "relative" part of this restriction which allows the next model state to depend on the current physical state. In that way, a linked list traversal can be supported by exposing *e.g.* the number of used heap locations in the refinement relation.

### Extending the wait-freedom verification

One immediate extension to our solution is verifying *bounded* wait-freedom, *i.e.*, termination within a fixed number of steps. In fact, all the case studies we consider in the paper, except the higher-order list mapping function, are bounded wait-free, but we only establish unbounded wait-freedom for them. Lawyer already supports proving bounded termination (see Section 4.7), and this should extend to the wait-freedom setting.

Another extension that should be done at the Trillium level is restoring the support for prophecy variables of Iris. This mechanism is needed for proving logically atomic triples for case studies such as the one in Section 5.5; these triples are known [13] to imply linearizability. With that, it will be possible to verify both wait-freedom and linearizability in the same framework. Moreover, the resulting logically atomic specification will be very useful for verifying client programs that use the wait-free operation.

### Verifying more properties

It is interesting to apply the obligations-based approach for verifying properties of non-terminating programs. In particular, it seems that proving "always-eventually $P$" properties can be supported in our framework by generating more fuel whenever $P$ is satisfied. It is worth investigating whether a larger subset of LTL properties can be supported with such extensions.

Another direction is combining liveness properties with trace safety properties. It seems that, at least when these properties are independent of each other, one can define a global Trillium model by composing separate models accounting for safety and liveness; the logic rules for taking a step with such a model should split into taking a step in each of these models independently too.

### Verifying termination in different settings

One direction for generalizing our framework is considering weak memory models. For examples such as the one in Figure 1.1a, it is important not only when *threads* write to memory, but also when the *system* makes these writes visible to other threads. This question is addressed by Lahav et al. [74] that extends various memory models

with the notion of fairness. To express this "memory fairness" in the logic, one can assign obligations not only to threads, but also to the system itself. A similar approach may apply to verifying termination in distributed systems [47].

Another direction is to consider the code that schedules the execution of individual threads. As with most of Iris-based projects, we consider a language with non-deterministic operational semantics where the next execution step can be taken by any active thread. In that setting, we simply assume fairness of the execution traces under consideration (see *e.g.* Definition 4 and Lemma 2). In real systems, however, the thread taking the next step is picked by the *scheduler*, which is a program by itself. Thus, to make our approach more realistic, we can verify the composition of the "user-space" programs and the scheduler; the latter always eventually choosing every active thread should imply the fairness of the execution. In a similar line of work, one can consider verifying cooperative concurrency, where threads explicitly yield to the scheduler.

# Part II

# Publications

# Chapter 4

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

**Abstract**

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.

## 4.1 Introduction

An important aspect of program verification frameworks [11, 40, 54, 62, 75, 80, 110] is *modularity*: that different modules of a program can be verified in isolation with respect to their individual 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 [60] used to achieve modularity when reasoning about the safety of concurrent programs is not generally sound for reasoning about liveness properties [9]. Second, the step-indexing technique [10] used to model higher-order concurrent program logics that support highly-modular specifications [63] is inherently at odds with liveness reasoning [49, 112, 118]. 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 [28, 118] 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 [60] 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 [62] — 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 used logic 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 [63].

To see the need for modularity and impredicative invariants, consider the example in Listing 4.1. 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 ex-

Listing 4.1: Example program

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

actly the same properties as $f$. This can be stated in terms of Hoare-style specifications as follows:

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

The specification (runAsync-spec) above quantifies over all possible functions $f$, all propositions $P$, and all predicates $\Phi$, stating that `runAsync` $f$ satisfies exactly the same Hoare triple as $f$ itself. (The postconditions 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 $\Phi(\mathrm{v})$ should hold, where $\Phi$ is an arbitrary predicate picked by the user of the `runAsync` module (and $\Phi$ could potentially involve invariants itself).

This expressivity of state-of-the-art program logics [11, 62] 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 [49]; 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. [118] 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. [118]. 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 [28], 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 [96], 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 [103, 104] 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* [15, 26, 52, 68, 76, 77, 104] that denote progress dependencies. Namely, a thread $\tau$ is allowed to busy-wait for a thread $\tau'$ if all obligations held by $\tau'$ are smaller than those held by $\tau$.

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 [28] 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 [103, 104]. To date, the soundness thereof has not been mechanized.

In this paper, we present Lawyer, which combines the obligations-based reasoning of Sassy [28] with the relational approach of Trillium [118]. 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 [28], 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 [95]. For a more detailed comparison of Sassy and Lawyer, including the challenges involved in incorporating Sassy's approach with Trillium's, see Sections 4.4 and 4.5.

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 (Section 4.2), 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 (Section 4.3).

- Case studies of modular verification of termination, including specification and proof of a fair-lock module and a client thereof (Section 4.4). 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 [95].

We discuss further related work in Section 4.5 and conclude in Section 4.6.

## 4.2 Obligations Model

In this section, we define the states and transitions of the 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 [80], 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 Section 4.2 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 Section 4.2 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 Section 4.2 and Section 4.2 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 Section 4.2 we give the formal definition of the refinement relation between program execution traces and OM traces. Moreover, in [95] we describe some additional properties that OM exhibits.

### No-Obligations Model Definition

The formal definition of NOM is given in Figure 4.1 on Page 40. The model NOM is parameterized by a finite partially ordered set of degrees $(\mathsf{Degree}, \preceq_{\mathsf{D}})$, a set $\mathsf{Thread}$ (to be instantiated with the set of thread identifiers of a given language) and a natural number $\mathsf{SB}$; we will describe these later on. The model states $\delta : \mathtt{NOM}_{\mathsf{St}}$ and transitions $\delta_1 \xrightarrow[\mathtt{NOM}]{\tau:\mathsf{Thread}} \delta_2$ 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 $\tau$ corresponds to an execution step of thread $\tau$.

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

A state of NOM is a record of type:

$$\left\{\begin{array}{l}\texttt{fuel} : \mathit{FinMultiset}(\mathsf{Barrel});\\ \texttt{phs} : \mathsf{Thread} \xrightarrow{\text{fin}} \mathsf{Phase};\\ \texttt{eb} : \mathbb{N}\end{array}\right\}$$

where

$$\begin{array}{rl}\mathsf{Barrel} & \triangleq \mathsf{Phase} \times \mathsf{Degree}\\ \mathsf{Phase} & \triangleq \mathit{List}\ \mathbb{N}\\ \pi_1 \preceq_{\mathsf{P}} \pi_2 & \triangleq \exists \pi'.\ \pi_2 = \pi' +\!\!+ \pi_1\end{array}$$

**Small transitions of NOM**

$$\text{NOMS-EXCHANGES-FUEL}\quad \frac{(\pi,d) \in \delta.\texttt{fuel} \qquad d' \prec_{\mathsf{D}} d \qquad n \leq \delta.\texttt{eb}}{\delta \underset{\texttt{NOM}}{\dashrightarrow} \delta[\texttt{fuel} := \delta.\texttt{fuel} \setminus \{1\cdot(\pi,d)\} \uplus \{n\cdot(\pi,d')\}]}$$

$$\text{NOMS-INCREASES-EB}\quad \frac{}{\delta \underset{\texttt{NOM}}{\dashrightarrow} \delta[\texttt{eb} := \delta.\texttt{eb}+1]}$$

**Fuel burning and the full transition of NOM**

$$\text{NOM-BURNS-FUEL}\quad \frac{\delta.\texttt{phs}[\tau] = \pi_\tau \qquad \pi \preceq_{\mathsf{P}} \pi_\tau \qquad (\pi,d) \in \delta.\texttt{fuel}}{\delta \overset{\tau}{\underset{\texttt{NOM}}{\rightsquigarrow}} \delta[\texttt{fuel} := \delta.\texttt{fuel} \setminus \{1\cdot(\pi,d)\}]}$$

$$\delta_1 \overset{\tau}{\underset{\texttt{NOM}}{\rightarrow}} \delta_2 \triangleq \exists n \leq \mathsf{SB}, \delta'.\ \delta_1 \underset{\texttt{NOM}}{\dashrightarrow}^{n} \delta' \wedge \delta' \overset{\tau}{\underset{\texttt{NOM}}{\rightsquigarrow}} \delta_2$$

Figure 4.1: No-Obligations model (parameterized by Degree, Thread and SB)

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 4.2. 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 $\mathsf{Degree} \triangleq \{d_0\}$[1] and $\mathsf{SB} \triangleq 0$. For simplicity, until Section 4.2 we also assume

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

Figure 4.2: Statically-known Bound

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

that $\mathsf{Thread} \triangleq \mathbb{N}$.

For simple examples such as Figure 4.2, 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

$$\left\{ \texttt{fuel} := \biguplus_{\tau \in T} \{N_\tau \cdot ([\tau], d_0)\}; \texttt{phs} := \lambda\, \tau \Rightarrow [\tau]; \texttt{eb} := 0 \right\}$$

for some choices of $N_\tau$, 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 state of the example program in Figure 4.2 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 $\overset{\tau}{\underset{\text{NOM}}{\rightsquigarrow}}$ (see Figure 4.1). Hence, according to NOM-BURNS-FUEL, a $\overset{\tau}{\underset{\text{NOM}}{\rightsquigarrow}}$ transition consumes one barrel assigned to $\tau$. With $\underset{\text{NOM}}{\rightsquigarrow}$ 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 $\underset{\text{NOM}}{\rightsquigarrow}$ and $\underset{\text{NOM}}{\rightarrow}$ 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 Section 4.3. 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 Section 4.2), 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 4.3, where `nondet` is a function which returns an arbitrary natural number, and `decr` is as in Figure 4.2. 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,

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

Figure 4.3: Dynamic Bound

we cannot directly repeat the process used for Figure 4.2 as the number of barrels to provide initially cannot be determined statically.

The key idea for verifying Figure 4.3 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 $\mathsf{Degree} \triangleq \{d_0, d_1, d_2\}$ such that $d_0 \prec_{\mathsf{D}} d_1 \prec_{\mathsf{D}} d_2$, and let $\mathsf{SB} \triangleq 1$; moreover, let $\tau \triangleq 0$ denote the only thread present in the program. Importantly, if the relation Degree on $\prec_{\mathsf{D}}$ is well-founded, the multiset $\{n \cdot d_0\}$ is smaller than $\{d_1\}$ for any $n$ [22].

We defer verifying `nondet` until Section 4.2 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* $\xrightarrow[\texttt{NOM}]{}$. 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 4.4 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 4.4 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 4.4), 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 ()
```

Figure 4.4: 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 this success in performing the **CAS** operation may necessitate an extra recursive call [52]. 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 [80] 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 4.7 in Section 4.2, 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 [103, 104] which allow sharing fuel *among threads with a shared ancestor*; in Section 4.2, 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 Section 4.2; in short, the main thread of the program initially has phase *nil*, and when a thread with phase $\pi_\tau$ 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 $\tau$ with a phase $\pi_\tau$ takes a step, we require fuel of some phase $\pi$ to be burned such that $\pi \preceq_{\mathsf{P}} \pi_\tau$ (the *phase order* $\preceq_{\mathsf{P}}$ is defined in Figure 4.1). Thus, if $\tau$ is provided with some fuel of phase $\pi$, it can either consume it itself or share it with any other thread $\tau'$ with phase $\pi_{\tau'}$ such that $\pi \preceq_{\mathsf{P}} \pi_{\tau'}$.

For Figure 4.4, in our Rocq development we choose $\mathsf{Degree} \triangleq \{d_0, d_1\}$ and show that a single call to `incr` can be verified separately by providing it with $2 \cdot (\pi, d_1)$. Here, $\pi$ 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 $\tau_l$ and $\tau_r$ 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 4.4.

Note that neither thread in Figure 4.4 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 Section 4.6. 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 4.3 where we assumed that $\texttt{eb} > k$ holds after nondet evaluates to some $k$. We verify nondet in Section 4.2; right now, the most important part is that it is implemented by repeatedly increasing a counter (see incr_loop in Figure 4.6). 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 $\texttt{eb} > k$ which is needed to verify Figure 4.3.

## 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 4.5 on Page 45. Besides Degree, Thread and SB, the model is parameterized by a finite partially ordered set of levels $(\mathsf{Level}, \preceq_{\mathsf{L}})$. The transition $\underset{\texttt{OM}}{\rightarrow}$ of OM is defined similarly to $\underset{\texttt{NOM}}{\rightarrow}$, 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

```
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
```

Figure 4.6: Blocking example

Consider the example in Figure 4.6. 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 $\tau_r$ busy-waits for the left thread $\tau_l$ 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.

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

A state of OM is a well-formed (see [95]) record of type $\{\texttt{fuel};\texttt{sigs};\texttt{obls};\texttt{eps};\texttt{phs};\texttt{eb}\}$ with fields' types as follows:

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

**Small transitions of** OM

OMS-EXCHANGES-FUEL

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

OMS-INCREASES-EB

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

OMS-CREATES-SIGNAL

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

OMS-SETS-SIGNAL

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

OMS-CREATES-EP

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

OMS-EXPECTS-EP

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

**Fuel burning, fork and the full transition of** OM

OM-BURNS-FUEL

$$\frac{\begin{array}{c}\delta.\texttt{phs}[\tau] = \pi_\tau \qquad \pi \preceq_{\mathsf{P}} \pi_\tau \\ (\pi,d) \in \delta.\texttt{fuel}\end{array}}{\delta \xrightarrow[\texttt{OM}]{\tau} \delta[\texttt{fuel} := \delta.\texttt{fuel} \setminus \{1 \cdot (\pi,d)\}]}$$

OM-FORKS

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

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

Figure 4.5: Obligations model (parameterized by Level, Degree, Thread and SB)

program in Figure 4.6, we instantiate OM with SB $\triangleq$ 2, Level $\triangleq \{\ell_0\}$, Degree $\triangleq \{d_0\}$. In our example, with a single busy-waiting loop, and hence a single obligation level, $\ell_0$, the partial order on Level is trivial, but more complicated examples can be supported by choosing an appropriate well-founded $\prec_{\mathsf{L}}$ 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

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

where $s_f \triangleq 0 : \mathsf{SignalId}$ is a *signal*, and the obligation of thread $\tau_l$ is 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 $\ell_0$ (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 $\tau_l$ has the obligation to set $s_f$ and, enforced by an invariant, must therefore set `f` eventually. If it did not, thread $\tau_l$ 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 4.6 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 [103, 104], 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 4.6, we take two OMS-EXPECTS-EP transitions every time the right thread, $\tau_r$, 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 [103, 104]). The side-condition on

levels, *i.e.*, $\forall o \in \texttt{obls}[\tau].\ l \prec_{\mathsf{L}} \delta.\texttt{sigs}[o].1$, in OMS-EXPECTS-EP rules out potential circular waiting among obligations which is crucial for guaranteeing termination — note that in Lemma 3 we require $\preceq_{\mathsf{L}}$ to be well-founded. In case of Figure 4.6, this side-condition holds trivially since $\tau_r$ has no obligations.

### Forking Threads

We now consider the implementation of `nondet` in Figure 4.7. 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 $\tau$ that forks the child thread $\tau'$ is matched by a forking transition, $\xrightarrow[\texttt{OM}]{\tau,\tau'}$. As can be seen in $\xrightarrow[\texttt{OM}]{\tau}$, this kind of transition can be taken after burning fuel. Forking transitions update the current phase of $\tau$ and assign the forked thread $\tau'$ 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
  fork (incr_loop f cnt);
  f := true; !cnt
```

Figure 4.7: Fork example. `incr_loop` is from *Figure* 4.6

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 4.7 follows closely that of Figure 4.6.

Note that without the phases (*i.e.* allowing unrestricted fuel sharing) we could (unsoundly) verify a variation of Figure 4.7 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.

## 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 $i$th state and label, respectively. In general, the traces we work with may be finite or infinite, but always non-empty. If $tr$ is finite, $tr.\mathsf{last}$ denotes its last state. Then, $tr :\langle \ell \rangle: s$ denotes the trace $tr$ extended with the transition from $tr.\mathsf{last}$ to the state $s$ with the label $\ell$ — this notation, of course, only makes sense, and is used when $tr$ is a finite trace. We write $\mathsf{dom}(tr)$ 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 $\mathscr{M}$ is *valid* if it denotes a sequence of $\mathscr{M}$ transitions:

$$valid_{\mathscr{M}}(tr) \triangleq \forall i.\, i+1 \in \mathsf{dom}\, tr \Rightarrow tr[i] \xrightarrow[\mathscr{M}]{tr\langle i\rangle} tr[i+1]$$

### Operational Semantics of $\lambda_{\mathsf{ref}}^{\mathsf{conc}}$

The language $\lambda_{\mathsf{ref}}^{\mathsf{conc}}$ is an ML-like programming language with higher-order references and concurrency. An excerpt of its operational semantics is shown in Figure 4.8; 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 $\lambda_{\mathsf{ref}}^{\mathsf{conc}}$ and traces of `OM`, we first define a relation $\simeq$ between the configurations of $\lambda_{\mathsf{ref}}^{\mathsf{conc}}$ 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 $\delta$, written $c \simeq \delta$, is defined as follows:*

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

**Syntax**

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

**Configuration reductions (with $\rightsquigarrow \subseteq (\mathit{Expr} \times \mathit{Heap})^2$ omitted; see [91])**

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

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

Figure 4.8: An excerpt of operational semantics of the language $\lambda_{\mathsf{ref}}^{\mathsf{conc}}$

*where*

$$
\begin{aligned}
\mathit{sameThreads}(c, \delta) &\triangleq \mathsf{dom}\, c.1 = \mathsf{dom}\, \delta.\texttt{obls} = \mathsf{dom}\, \delta.\texttt{phs} \\
\mathit{liveThreads}(c, \delta) &\triangleq \forall \tau.\, \mathit{hasObls}(\tau, \delta) \Rightarrow \mathit{threadActive}(\tau, c) \\
\mathit{threadActive}(\tau, c) &\triangleq c.1[\tau] \notin \mathit{Val} \\
\mathit{hasObls}(\tau, \delta) &\triangleq \delta.\texttt{obls}[\tau] \neq \emptyset
\end{aligned}
$$

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 $\lambda_{\mathsf{ref}}^{\mathsf{conc}}$ and let mtr be a finite* `OM` *trace. The refinement relation between etr and mtr, written $\xi_{\texttt{OM}}^{\Lambda}(\mathit{etr}, \mathit{mtr})$ is defined as follows:*

$$
\begin{aligned}
\xi_{\texttt{OM}}^{\Lambda}(\mathit{etr}, \mathit{mtr}) \triangleq\ & \mathit{valid}_{\lambda_{\mathsf{ref}}^{\mathsf{conc}}}(\mathit{etr}) \wedge \mathit{valid}_{\texttt{OM}}(\mathit{mtr}) \wedge \mathsf{dom}\, \mathit{etr} = \mathsf{dom}\, \mathit{mtr} \wedge \\
& (\forall i \in \mathsf{dom}\, \mathit{etr}.\, \mathit{etr}[i] \simeq \mathit{mtr}[i]) \wedge \\
& (\forall i.\, i + 1 \in \mathsf{dom}\, \mathit{etr} \Rightarrow \mathit{etr}\langle i \rangle = \mathit{mtr}\langle i \rangle)
\end{aligned}
$$

In order to instantiate Trillium with the refinement relation $\xi_{\texttt{OM}}^{\Lambda}$, we need to show it is relatively image-finite [118].

**Lemma 1.** *$\xi^{\Lambda}_{\mathtt{OM}}$ is relative image-finite:*

$$\forall etr, \tau_1, c, mtr.\ \mathsf{finite}\big\{(\tau_2, \delta) \mid \xi^{\Lambda}_{\mathtt{OM}}(etr : \langle \tau_1 \rangle : c, mtr : \langle \tau_2 \rangle : \delta)\big\}$$

**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 [95]) 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). Then, we show that any fair trace *mtr* of `OM` is necessarily finite (Lemma 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 $\mathscr{M}$ and an enabledness predicate* $\mathtt{E} : \mathtt{Label}(\mathscr{M}) \to \mathtt{State}(\mathscr{M}) \to$ *Prop, a trace tr of $\mathscr{M}$ is fair with respect to* `E`*, written $FairTrace_{\mathtt{E}}(tr)$, iff:*

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

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

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

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

The definition of $\mathit{FairTrace}_{\mathtt{E}}$ 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, $\mathit{fairExec}(\mathit{etr})$ for a finite trace $\mathit{etr}$ implies that all threads have terminated.

**Lemma 2.** $\forall \mathit{etr}, \mathit{mtr}.\ \mathit{etr} \hat{\preccurlyeq}_{\xi^{\wedge}_{\mathtt{OM}}} \mathit{mtr} \wedge \mathit{fairExec}(\mathit{etr}) \Rightarrow \mathit{oblsFair}(\mathit{mtr})$.

**Lemma 3.** *If both* $\prec_{\mathsf{L}}$ *and* $\prec_{\mathsf{D}}$ *are well-founded, then*

$$\forall \mathit{mtr}.\ \mathit{oblsFair}(\mathit{mtr}) \Rightarrow \mathsf{finite}(\mathit{mtr})$$

## 4.3 Program Logic

As explained in Section 4.2, 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 3. To this end, we have developed the Lawyer program logic (Hoare logic) on top of Trillium's [118] 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 $\{P\}\ e\ \{\Phi\}^{\tau}_{\mathscr{E}}$. Intuitively, $\{P\}\ e\ \{\Phi\}^{\tau}_{\mathscr{E}}$ means that, given that the precondition $P$ holds, the program $e$ is safe to run (*i.e.* without getting stuck) under thread $\tau$, 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 $\{P\}\ e\ \{\Phi\}^{\tau}_{\mathscr{E}}$ only invariants whose names appear in the mask $\mathscr{E}$ can be accessed. Whenever all invariants are accessible, *i.e.*, the mask $\mathscr{E}$ 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 4.7 (Page 47). 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:

- There are at least two fuel degrees $d_0, d_1 : \mathsf{Degree}$ such that $d_0 \prec_{\mathsf{D}} d_1$.
- There is at least one obligation level $\ell_0 : \mathsf{Level}$.
- Every `OM` transition allows at least 5 small transitions, *i.e.* $5 \leq \mathsf{SB}$.

**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 composing the modules' specifications: the restrictions placed by each of

| Ownership of the resource … | implies … for $\delta$, the current state of OM |
|---|---|
| $\mathsf{bar}\,(\pi : \mathsf{Phase})\,(d : \mathsf{Degree})$ | $\exists \pi_0.\ \{1 \cdot (\pi_0, d)\} \subseteq \delta.\mathtt{fuel} \wedge \pi_0 \preceq_{\mathsf{P}} \pi$ |
| $\mathsf{sgn}\,(s : \mathsf{SignalId})\,(\ell : \mathsf{Level})\,(b : \mathbb{B})$ | $\delta.\mathtt{sigs}[s] = (\ell, b)$ |
| $\mathsf{sgn}_{\exists}\,(s : \mathsf{SignalId})\,(\ell : \mathsf{Level})$ | $\exists b.\ \delta.\mathtt{sigs}[s] = (\ell, b)$ □ |
| $\mathsf{obls}\,(\tau : \mathsf{Thread})\,(R : \mathit{FinSet}(\mathsf{SignalId}))$ | $\delta.\mathtt{obls}[\tau] = R$ |
| $\mathsf{ep}\,(s : \mathsf{SignalId})\,(\pi : \mathsf{Phase})\,(d : \mathsf{Degree})$ | $\exists \pi_0.\ (s, \pi_0, d) \in \delta.\mathtt{eps} \wedge \pi_0 \preceq_{\mathsf{P}} \pi$ □ |
| $\mathsf{ph}_{(q:\mathbb{Q}) \in (0..1]}\,(\tau : \mathsf{Thread})\,(\pi : \mathsf{Phase})$ | $\delta.\mathtt{phs}[\tau] = \pi$ |
| $\mathsf{elb}\,(n : \mathbb{N})$ | $n \leq \delta.\mathtt{eb}$ □ |

Abbreviations:

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

Figure 4.9: Resources of the Lawyer logic. The □ marker denotes persistent resources.

them are combined appropriately to obtain a specification for the composition. For example, Section 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 $\tau$ and phase $\pi$:

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

Intuitively, the precondition states (see Figure 4.9) that nondet can be called when the state $\delta$ of OM satisfies the following properties: thread $\tau$ holds no obligations and has phase $\pi$ assigned to it, and that phase $\pi$ 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, that it refines OM, and that when it terminates, it returns $n \in \mathbb{N}$ and ends in some state $\delta'$ of OM such that the postcondition holds. The postcondition intuitively states that in the state $\delta'$ the exchange bound is at least $n$, written as the proposition $\mathsf{elb}\,n$, and that the thread $\tau$ still has no obligations, but its phase is now increased to $\pi'$. 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 4.3.

Intuitively, nondet terminates because the main thread sets the flag telling the forked thread to stop incrementing the counter. As explained in Section 4.2, 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*:

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

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]

INV-OPEN-SIMPL
$$\{R * P\}\, e\, \{Q * P\}^{\tau}_{\mathscr{E}\setminus\mathcal{N}} \wedge e \text{ is atomic} \wedge \mathcal{N} \subseteq \mathscr{E} \vdash \left\{R * \boxed{P}^{\mathcal{N}}\right\} e\, \{Q\}^{\tau}_{\mathscr{E}}$$

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.

## Establishing the Invariant

After the first two lines of the `nondet` example (Figure 4.7), we obtain $\mathit{flag} \mapsto$ **`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: $\{P\}\, e\, \{v.\, Q\}^{\tau}_{\mathscr{E}} \triangleq \Box\left(P \mathrel{-\!\!*} \mathsf{wp}^{\tau}_{\mathscr{E}}\, e\, \{v.\, Q\}\right)$ — 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. [63] for details. For those familiar with Iris, we remark that all invariants we formally state in the paper are timeless anyway.

$$\frac{}{\mathsf{OU}\,\{\mathsf{elb}\,0\}}\ \text{OU-EB-0} \qquad \frac{\mathsf{bar}\,\pi\,d \qquad \mathsf{elb}\,b \qquad d' \prec_{\mathsf{D}} d}{\mathsf{OU}\,\{b \cdot \mathsf{bar}\,\pi\,d'\}}\ \text{OU-EXCHANGES-FUEL}$$

$$\frac{\mathsf{obls}\,\tau\,R}{\mathsf{OU}\,\{\exists s \notin R.\ \mathsf{sgn}\,s\,\ell\,\mathsf{False} * \mathsf{obls}\,\tau\,R \cup \{s\}\}}\ \text{OU-CREATES-SIGNAL} \qquad \frac{\mathsf{obls}\,\tau\,R \qquad s \in R \qquad \mathsf{sgn}\,s\,\ell\,(b : \mathbb{B})}{\mathsf{OU}\,\{\mathsf{sgn}\,s\,\ell\,\mathsf{True} * \mathsf{obls}\,\tau\,R \setminus \{s\}\}}\ \text{OU-SETS-SIGNAL}$$

$$\frac{\mathsf{bar}\,\pi\,d \qquad \mathsf{sgn}_{\exists}\,s\,\ell \qquad \mathsf{ph}_q\,\tau\,\pi \qquad d' \prec_{\mathsf{D}} d}{\mathsf{OU}\,\{\mathsf{ep}\,s\,\pi\,d' * \mathsf{ph}_q\,\tau\,\pi\}}\ \text{OU-CREATES-EP} \qquad \frac{\mathsf{elb}\,b}{\mathsf{OU}\,\{\mathsf{elb}\,(b+1)\}}\ \text{OU-INCREASES-EB}$$

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

Figure 4.10: Rules for the obligation update modality OU

Following Hinrichsen et al. [47], we decompose weakest preconditions into two parts: a *single-step* weakest precondition [82] followed by a *model update* modality:

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

Single-step weakest preconditions are like weakest preconditions except they demand their postcondition 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:

$$\text{MU-OM-SIMPL}\quad \mathsf{OU}^{i}\,\{P * \mathsf{ph}\,\tau\,\pi * \mathsf{bar}\,\pi\,d\} \land i \leq \mathsf{SB} \vdash \mathsf{MU}^{\tau}\{P * \mathsf{ph}\,\tau\,\pi\}$$

where $\mathsf{OU}^{i}\,\{P\}$ is $i$ times repeated obligation update modalities: $\mathsf{OU}^{0}\,\{P\} \triangleq P$ 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 Section 4.4.) The rules for deriving obligation updates are given in Figure 4.10. Using these rules, we can derive

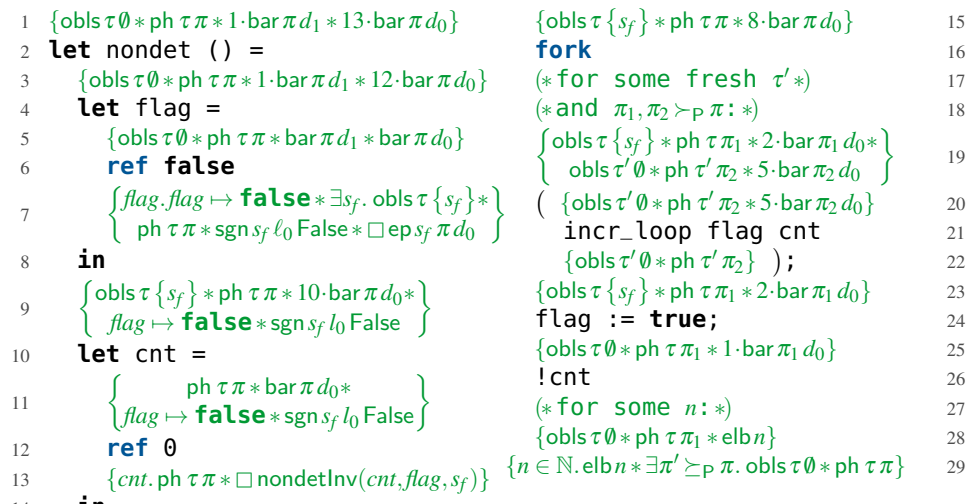



Figure 4.11: Proof of `nondet` specification

the following

$$\text{NONDET-OU-UPD}\quad \frac{\mathsf{obls}\,\tau\,\emptyset \qquad \mathsf{ph}\,\tau\,\pi \qquad \mathsf{bar}\,\pi\,d_1}{\mathsf{OU}\{\mathsf{OU}\{\exists s_f.\ \mathsf{obls}\,\tau\,\{s_f\} * \mathsf{ph}\,\tau\,\pi * \mathsf{sgn}\,s_f\,\ell_0\,\mathsf{False} * \mathsf{ep}\,s_f\,\pi\,d_0\}\}}$$

where $\mathsf{ep}\,s\,\pi\,d$ asserts existence of an expectation permission for signal $s$, allowing one to generate fuel at degree $d$ and phase of at least $\pi$. 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).

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

## Verifying `nondet`'s Main Thread

The proof of the `nondet` specification (`nondet`-spec) is sketched in Figure 4.11. 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:

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

This is a special derived rule for the `nondet` example; the general version is shown in [95]. We split the available resources to verify the two threads separately. The verification of the forked thread is described below in Section 4.3. Notice that postconditions of both threads in Figure 4.11 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 proving 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 24 for which we prove the following, which requires accessing the invariant using the rule INV-OPEN-SIMPL:

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

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

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

Finally, we read the counter on line 26, again accessing the invariant using the rule INV-OPEN-SIMPL to obtain the points-to and 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:

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

### Verifying `incr_loop`

Figure 4.12 sketches the proof of the `incr_loop` specification used in Figure 4.11 (lines 20-22). Note the recursive call on line 11. Following the standard way to verify

```
1  { obls τ ∅ ∗ ph τ π ∗ 5·bar π d0 ∗
     □(ep sf π d0 ∗ nondetInv(cnt, flag, sf)) }
2  let rec incr_loop flag cnt =
3    {obls τ ∅ ∗ ph τ π ∗ 4·bar π d0}
4    let t := !flag in
5    { obls τ ∅ ∗ ph τ π ∗ 2·bar π d0 ∗
       (t = False ⇒ 5·bar π d0) }
6    if t then () {obls τ ∅ ∗ ph τ π}
7    else
8      {obls τ ∅ ∗ ph τ π ∗ (1 + 5)·bar π d0}
9      FAA(cnt, 1);
10     {obls τ ∅ ∗ ph τ π ∗ 5·bar π d0}
11     incr_loop flag cnt
12     {obls τ ∅ ∗ ph τ π}
13 {obls τ ∅ ∗ ph τ π}
```

Figure 4.12: Proof sketch for `incr_loop`

recursive functions [12], 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 physical step and framing the irrelevant resources, we prove the following model update using MU-OM-SIMPL choosing $i$ depending on $b$:

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

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.

## 4.4 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 4.13, 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

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

Figure 4.13: Client using the fair lock and ticket lock implementation

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 4.13 terminates is that every lock acquisition is followed by a release. Lawyer's mechanism for specifying such requirements, as we saw in Section 4.2 and Section 4.3, 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 the lock, *i.e.*, when `FAA` at line 7 succeeds, we can assign that thread an obligation to release the lock. This obligation is fulfilled upon the write at line 4. On the other hand, when a thread fails to acquire the lock (*i.e.*, the check at line 10 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 Section 4.3 and assign the left thread an obligation, which it fulfills upon setting the `flag`.

$$\text{BOU-INTRO}\ \frac{P}{\mathsf{BOU}_n^{\mathscr{E}}\{P\}} \qquad \text{BOU-WAND}\ \frac{P \mathrel{-\!\!*} Q}{\mathsf{BOU}_n^{\mathscr{E}}\{P\} \mathrel{-\!\!*} \mathsf{BOU}_n^{\mathscr{E}}\{Q\}} \qquad \text{BOU-OU}\ \frac{\mathsf{OU}\{\mathsf{BOU}_n^{\mathscr{E}}\{P\}\}}{\mathsf{BOU}_{n+1}^{\mathscr{E}}\{P\}}$$

$$\text{BOU-SPLIT}\ \frac{\mathsf{BOU}_n^{\mathscr{E}}\{\mathsf{BOU}_m^{\mathscr{E}}\{P\}\}}{\mathsf{BOU}_{n+m}^{\mathscr{E}}\{P\}} \qquad \text{BOU-WEAKEN}\ \frac{P_1 \mathrel{-\!\!*} P_2 \qquad n_1 \le n_2 \qquad \mathscr{E}_1 \subseteq \mathscr{E}_2 \qquad \mathsf{BOU}_{n_1}^{\mathscr{E}_1}\{P_1\}}{\mathsf{BOU}_{n_2}^{\mathscr{E}_2}\{P_2\}}$$

$$\text{BOU-INVS}\ \frac{{}_{\mathscr{E}}\Rrightarrow_{\emptyset} \mathsf{BOU}_n^{\emptyset}\{{}_{\emptyset}\Rrightarrow_{\mathscr{E}} P\}}{\mathsf{BOU}_n^{\mathscr{E}}\{P\}}$$

Figure 4.14: Selected rules for BOU

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 (Section 4.4) that allows us to achieve better flexibility in modular specifications. Afterwards, in Section 4.4 we show how to use this modality to give the fair lock specification, and discuss how the ticket lock implementation satisfies it in Section 4.4. Finally, in Section 4.4 we briefly explain how to verify `client` on top of this fair lock specification.

## Bounded Obligations Update Modality

Recall that the OU modality presented in Section 4.3 represents exactly one small `OM` transition. We now introduce a more general variant of this modality, called the "bounded obligations update" modality, $\mathsf{BOU}_n^{\mathscr{E}}\{P\}$, which intuitively expresses that $P$ holds after taking up to $n$ small transitions and using invariants in $\mathscr{E}$ — we call $n$ the "size" of $\mathsf{BOU}_n^{\mathscr{E}}\{P\}$. Figure 4.14 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 proofs when performing a model update:

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

In particular, the rule MU-OM-SIMPL we had shown earlier in Section 4.3 can be derived from MU-OM-NOFORK.

$$\forall \tau, \pi, u, P.\ \begin{array}{l}\{P * \mathscr{B}_{\mathrm{FL}}(u) \leq \mathsf{SB} * \mathsf{bar}\,\pi\, d_{\mathrm{FL}} * \mathsf{ph}\,\tau\,\pi\} \\ \quad \texttt{newLock}\ () \\ \left\{lk.\ \mathsf{FairLock}^{lk}(u,P) * \mathsf{ph}\,\tau\,\pi\right\}^{\tau}\end{array} \qquad (\textit{newlockSpec})$$

where

$$\mathsf{FairLock}^{lk}(u,P) \triangleq \exists \mathsf{locked} : \mathsf{SignalId} \to \mathit{iProp},\ \iota_{\mathrm{FL}} : \mathscr{N}.$$

$$\square\,(\forall s_1, s_2.\ \mathsf{locked}(s_1) * \mathsf{locked}(s_2) \mathrel{-\!\!*} \mathsf{False}) \wedge \qquad (\text{lockedExcl})$$

$$\square\,(\forall s_r.\ \mathsf{locked}(s_r) \mathrel{-\!\!*} \exists l_r \in L_{\mathrm{FL}}.\ \mathsf{sgn}_{\exists}\, s_r\, l_r) \wedge \qquad (\textit{lockedLvl})$$

$$\left(\begin{array}{l}\forall \tau, \pi, \mathscr{O}. \\ \{\mathsf{bar}\,\pi\, d_{\mathrm{FL}} * \mathsf{ph}\,\tau\,\pi * \mathsf{obls}\,\tau\,\mathscr{O} * L_{\mathrm{FL}} \prec_{\mathsf{L}} \mathsf{sgn}\,\mathscr{O}\} \\ \quad \texttt{acquire}\ lk \\ \{\exists s_r \notin \mathscr{O}.\ P * \mathsf{locked}(s_r) * \mathsf{ph}\,\tau\,\pi * \mathsf{obls}\,\tau\,(\mathscr{O} \cup \{s_r\})\}^{\tau}\end{array}\right) \wedge \qquad (\textit{acquireSpec})$$

$$\left(\begin{array}{l}\forall \tau, \pi, \mathscr{O}, s_r, Q. \\ \left\{\begin{array}{c}\mathsf{locked}(s_r) * \mathsf{bar}\,\pi\, d_{\mathrm{FL}} * \mathsf{ph}\,\tau\,\pi * \mathsf{obls}\,\tau\,(\mathscr{O} \cup \{s_r\}) * \\ (\forall q.\ \mathsf{ph}_q\,\tau\,\pi \mathrel{-\!\!*} \mathsf{obls}\,\tau\,\mathscr{O} \mathrel{-\!\!*} \mathsf{BOU}_u^{\top \setminus \{\iota_{\mathrm{FL}}\}}\ \{P * (\mathsf{ph}_{1-q}\,\tau\,\pi \mathrel{-\!\!*} Q)\})\end{array}\right\} \\ \quad \texttt{release}\ lk \\ \{Q\}^{\tau}\end{array}\right) \qquad (\textit{releaseSpec})$$

Figure 4.15: Fair lock specification (parameterized by $L_{\mathrm{FL}} : \mathit{FinSet}(\mathsf{Level})$, $d_{\mathrm{FL}} : \mathsf{Degree}$ and $\mathscr{B}_{\mathrm{FL}} : \mathbb{N} \to \mathbb{N}$)

### Fair Lock Specification

Our specification for a fair lock is given in Figure 4.15. 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 [12]. 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 [12] is simply True. We emphasize that this lock specification in Figure 4.15, just like the one given by Birkedal and Bizjak [12], is abstract: it hides the implementation by

existentially quantifying over the predicate locked and the invariant name $\iota_{\mathtt{FL}}$.

The specification in Figure 4.15 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 $d_{\mathtt{FL}}$. 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 $\mathsf{locked}(s_r)$ token, while the release operation fulfills $s_r$ for which it receives $\mathsf{locked}(s_r)$. The level of the release obligation belongs to the set $L_{\mathtt{FL}}$. The lock client can carry other obligations while acquiring the lock, but the levels of these other obligations must be higher than those in $L_{\mathtt{FL}}$. Parameterization of Figure 4.15 allows imposing restrictions on *e.g.* obligations levels used by each individual lock instance created with `newLock`, which is needed to verify Figure 4.13 (see Section 4.4) and an example with multiple locks (provided in [95]).

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 Section 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 supporting 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* $\mathscr{B}_{\mathtt{FL}}(u)$ 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 $\mathscr{B}_{\mathtt{FL}}(u) \leq \mathsf{SB}$. The condition $\mathscr{B}_{\mathtt{FL}}(u) \leq \mathsf{SB}$ 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.

## Verifying the Ticket Lock Implementation

Here we discuss the proof that the ticket lock implementation satisfies the specification in Figure 4.15 (for any choice of parameters $L_{\mathtt{FL}}$, $d_{\mathtt{FL}}$ and $\mathscr{B}_{\mathtt{FL}}$ satisfying a number of assumptions wrt. `OM`; see [95]). For space reasons, we only briefly describe the proof from a high-level point of view. We refer the reader to the accompanying material [95] for further details.

[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.

The proof proceeds in two stages. First, we derive the fair lock specification in Figure 4.15 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 [95].

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 [62, 100] are used for safety reasoning in state-of-the-art program logics to support modular reasoning by internalizing the well-known idea of linearizability [46]. 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 [46] of the method in question. (For an introduction to such logically atomic specifications and proofs relying on them, see Birkedal and Bizjak [12, 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 [28], 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 [28] shows, supports verifying a number of challenging examples, including lock handoffs and implementing nested locks, *e.g.*, cohort locks [28]. 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 [28]. 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 4.15, which can be straightforwardly used to verify clients like that in Figure 4.13.

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 [76] 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.

### Verification of the Lock Client Program

To state and prove the specification of the `client` function in Figure 4.13, we assume a lock implementation that satisfies the specification in Figure 4.15 for some $d_{\mathtt{FL}}$, $L_{\mathtt{FL}}$ and $\mathscr{B}_{\mathtt{FL}}$. Moreover, we assume the following holds for the obligations model's parameters:

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

Note that these assumptions still don't fix the concrete choice for $\mathsf{Level}, \mathsf{Degree}$ and $\mathsf{SB}$ 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:

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

Due to space reasons, we relegate the details of the proof to our accompanying technical appendix [95].

## 4.5 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 [28], 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 [115] explain, truly modular separation logic specifications require *impredicativity*. Lilo [76] provides a mechanized program logic that abstracts the construction of thread-local simulation relations of [75]. 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 Section 4.4. 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 4.13, requiring to specify all modules' invariants in the initial state. Both Sassy and Lilo, as well as Lawyer, take inspiration from TaDA Live [26], 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 Section 4.6. Simuliris [33] 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 [80] verifies properties of blocking and non-blocking concurrent modules. Lili introduced the notions of "blocking" and "delay" (explained in Section 4.2) 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 [38, 40, 65]. In particular, Kim et al. [65] 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 [118] 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 [118], 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

[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.

modules as we did here for fair locks in Section 4.4. Tassarotti et al. [117] showed that Iris can be used to reason about fair termination-preserving refinement between two programs, which they used for proving the 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 [85, 112] support modular termination verification in an impredicative setting, but they restrict the programs they can verify. Transfinite Iris [112] 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 [85] uses Iris' [62] 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.

## 4.6 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 proving 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 4.4 and Figure 4.2 terminate even under an unfair scheduler. Moreover, we have also used a specific instantiation of `OM` to show the program in Figure 4.2 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 [95] for details. As a part of future work, we investigate the applicability of Lawyer to verifying lock-freedom of Figure 4.4 and wait-freedom of Figure 4.2.

**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 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 4.3 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 $\underset{\texttt{OM}}{\rightarrow}$. 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 [118, the `nondet_start` example].

## Data-Availability Statement

We provide the Rocq mechanization [95] of all results presented in this paper, including definition of the Obligations model and the refinement relation $\xi^{\Lambda}_{\texttt{OM}}$, Lawyer program logic and its adequacy theorem, and proofs of termination of all the examples and case studies (Figure 4.13 and a number of others).

## 4.7 Appendix

### Well-formedness of Obligations Model State

**Definition 5** (Well-formedness of $\texttt{OM}_{\texttt{St}}$)**.** *A state* $\delta : \texttt{OM}_{\texttt{St}}$ *is* well-formed *iff the following conditions hold:*

- $\mathsf{dom}\,\delta.\texttt{phs} = \mathsf{dom}\,\delta.\texttt{obls}$

- $\forall s, l.\ \ \delta.\mathtt{sigs}[s] = (l, \mathsf{False}) \Rightarrow \exists \tau.\, s \in \delta.\mathtt{obls}[\tau]$
- $\forall s, \tau.\ \ s \in \delta.\mathtt{obls}[\tau] \Rightarrow s \in \mathsf{dom}\,\delta.\mathtt{sigs}$
- $\forall \tau_1, \tau_2 \in \mathsf{dom}\,\delta.\,\mathtt{obls}.\ \delta.\mathtt{obls}[\tau_1] \,\#\, \delta.\mathtt{obls}[\tau_2]$
- $\forall \tau_1, \tau_2 \in \mathsf{dom}\,\delta.\,\mathtt{phs}.\ \neg(\delta.\mathtt{phs}[\tau_1] \preceq_{\mathsf{P}} \delta.\mathtt{phs}[\tau_2] \vee \delta.\mathtt{phs}[\tau_2] \preceq_{\mathsf{P}} \delta.\mathtt{phs}[\tau_1])$
- $\forall \tau \in \mathsf{dom}\,\delta.\,\mathtt{phs}, (\pi, _) \in \delta.\mathtt{fuel}.\ \neg(\delta.\mathtt{phs}[\tau] \prec_{\mathsf{P}} \pi)$
- $\forall \tau \in \mathsf{dom}\,\delta.\,\mathtt{phs}, (_, \pi, _) \in \delta.\mathtt{eps}.\ \neg(\delta.\mathtt{phs}[\tau] \prec_{\mathsf{P}} \pi)$

**Lemma 4** (Preservation of $\mathtt{OM}_{\mathsf{St}}$ well-formedness)**.** *For any $\delta_1 \xrightarrow[\mathtt{OM}]{\tau} \delta_2$, if $\delta_1$ is well-formed, so is $\delta_2$.*

*Proof.* Proven for burning and forking steps, as well as all cases of small transitions. □

**Definition 6.** *Let $c$ be a $\lambda^{\mathsf{conc}}_{\mathsf{ref}}$ configuration with the single thread $\tau$, and any $C$ : $FinMultiset(\mathsf{Barrel})$, there is a corresponding* initial state *of* OM*, written* $\mathtt{OM}initial_{\tau,C}$*, such that $c \simeq \mathtt{OM}initial_{\tau,C}$. This initial state is defined as follows:*

$$\mathtt{OM}initial_{\tau,C} \triangleq \left\{\begin{array}{c}\mathtt{fuel} := \{1 \cdot nil\} \times C; \mathtt{sigs} := \emptyset; \mathtt{obls} := \{\tau \mapsto \emptyset\}; \\ \mathtt{eps} := \emptyset; \mathtt{phs} := \{\tau \mapsto nil\}; \mathtt{eb} := 0\end{array}\right\}$$

**Lemma 5** (Well-formedness of initial $\mathtt{OM}_{\mathsf{St}}$ state)**.** $\mathtt{OM}initial_{\tau,C}$ *is well-formed for any $\tau$ and $C$.*

## Adequacy theorem of Lawyer

In this section, we provide a specialized version of the adequacy theorem of Lawyer. In the Rocq development, we generalize it, allowing multiple threads and a non-empty heap in the starting configuration, as well as relaxing restrictions on the starting state of OM.

**Theorem 3** (Lawyer adequacy)**.** *Let etr be a potentially infinite execution trace of $\lambda^{\mathsf{conc}}_{\mathsf{ref}}$. Assume that $etr[0] = ([\tau_0 \mapsto e], \emptyset)$ for some $e$ : Expr and the initial thread identifier $\tau_0$ : Thread. If the triple*

$$\left\{\exists \pi.\ \mathsf{obls}\,\tau_0\,\emptyset * \mathsf{ph}\,\tau_0\,\pi * \bigstar_{d \in D} \mathsf{bar}\,\pi\,d\right\}\ e\ \{\mathsf{obls}\,\tau\,\emptyset\}^{\tau_0}$$

*is derivable for some $D$ : $FinMultiset(\mathsf{Degree})$, then $fairExec(etr) \Rightarrow \mathsf{finite}(etr)$.*

### Other Properties of Obligations Model

In this section, we provide the other termination properties of OM applicable for certain choices of the construction parameters.

First, *oblsFair* doesn't have to be shown by transporting fairness from execution. Verification of non-blocking programs doesn't require generating fuel, which removes the need for obligations altogether and makes *oblsFair* hold trivially. This can be used to guarantee termination of non-blocking programs even under an unfair scheduler.

**Corollary 1.** *All traces of* OM *instantiated with* $\mathsf{Level} \triangleq \emptyset$ *are finite.*

*Proof.* By Lemma 3, we need to show that an arbitrary model trace $tr$ is obligations-fair. Consider a thread that holds an obligation somewhere in the trace: $s \in tr[i].\mathtt{obls}[\tau]$. In a well-formed obligations state, an obligation corresponds to an unset signal, so $\exists l \in \mathsf{Level}.\ tr[i].\mathtt{sigs}[s] = (l, \mathsf{False})$. However, it contradicts the emptiness of $\mathsf{Level}$. □

Then, without fuel generation and fuel exchanges, the length of OM traces is bounded by the initial amount of fuel.

**Corollary 2.** *Any trace mtr of* OM *instantiated with* $\mathsf{Level} \triangleq \emptyset, \mathsf{Degree} \triangleq ()$ *has a maximal length of* $|mtr[0].\mathtt{fuel}|$.

*Proof.* With such instantiation of OM, the only allowed small transition is OMS-INCREASES-EB which doesn't affect fuel. Therefore, the amount of available fuel will only decrease with every step of OM. □

### Operational semantics of the language $\lambda_{\mathsf{ref}}^{\mathsf{conc}}$

Figure 4.16 shows more details of $\lambda_{\mathsf{ref}}^{\mathsf{conc}}$'s operational semantics, including part of the per-thread reduction definition. We refer the reader to [95] for the full semantics of $\lambda_{\mathsf{ref}}^{\mathsf{conc}}$.

### Full Rules of Lawyer

See Figure 4.17.

### Total Atomic Updates

Our *total atomic updates* (inspired by those of [28]), or TAUs, are a liveness-aware extension of Iris' atomic updates [62]. The latter are used in logically atomic method specifications, allowing updating arbitrary resources at the linearization point. TAUs additionally allow taking OM steps before and at the linearization point, thus enabling liveness reasoning with logically atomic specifications. Below, we introduce the corresponding additions to the atomic updates in multiple steps, highlighting the changes in **green**.

**Syntax**

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

**Pure reductions** **Per-thread reductions**

$$
(\texttt{rec}\, f\, (x) := e)\, v \overset{\text{pure}}{\rightsquigarrow} e[\texttt{rec}\, f\, (x) := e/f][v/x] \qquad (e, h) \rightsquigarrow (e', h) \;\; (\text{if } e \overset{\text{pure}}{\rightsquigarrow} e')
$$

$$
\texttt{fst}(v_1, v_2) \overset{\text{pure}}{\rightsquigarrow} v_1 \qquad (!\,\ell, h[\ell \mapsto v]) \rightsquigarrow (v, h[\ell \mapsto v])
$$

$$
\vdots \qquad\qquad \vdots
$$

**Configuration reductions**

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

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

Figure 4.16: An excerpt of operational semantics of the language $\lambda^{\mathsf{conc}}_{\mathsf{ref}}$

Usual atomic updates look like the non-highlighted part of the notation below:

$$
{}^{b}_{\tau \mapsto \pi}\left\langle \vec{x}.\, P \mid \vec{y}.\, Q \Rrightarrow \Phi \right\rangle^{\mathcal{O}}_{\mathscr{E}} \tag{TAU-1}
$$

Here, $\vec{x}, \vec{y}$ are optional binders; $\vec{x}$ is free in $P$, and both $\vec{x}, \vec{y}$ are free in $Q$ and $\Phi$. The non-highlighted part of TAU-1 can be read as follows: by opening invariants in $\mathscr{E}$, one obtains $P$ for some witness $\vec{x}$ and can either:

- *abort*, *i.e.*give $P$ back to close the invariants as is, reestablishing the atomic update;
- *commit*, *i.e.*provide $Q$ for any chosen $\vec{y}$ and close the invariants, obtaining $\Phi$ for $\vec{x}, \vec{y}$. The update from $P$ to $Q$ corresponds to the execution of the linearization point.

$$\text{TRILLIUM-VAL}\quad \frac{\Phi(v) \quad v \in \mathit{Val}}{\mathsf{wp}^{\tau}_{\mathscr{E}}\, v\, \{\Phi\}} \qquad \text{TRILLIUM-STEP-FORK}\quad \frac{{}_{\mathscr{E}}\Rrightarrow_{\mathscr{E}'} \forall \tau'.\ \triangleright \mathsf{MU}^{\tau,\tau'}_{\mathscr{E}'} \{{}_{\mathscr{E}'}\Rrightarrow_{\mathscr{E}} \mathsf{wp}^{\tau'}\, e\, \{\mathsf{obls}\, \tau' \emptyset\} * \Phi\}}{\mathsf{wp}^{\tau}_{\mathscr{E}}\, \texttt{fork}(e)\, \{\Phi\}}$$

$$\text{TRILLIUM-STEP-NOFORK}\quad \frac{{}_{\mathscr{E}}\Rrightarrow_{\mathscr{E}'} \mathsf{sswp}_{\mathscr{E}'}\, e\, \{e'.\triangleright \mathsf{MU}^{\tau}_{\mathscr{E}'} \{{}_{\mathscr{E}'}\Rrightarrow_{\mathscr{E}} \mathsf{wp}^{\tau}_{\mathscr{E}}\, e'\, \{\Phi\}\}\} \qquad e \neq \texttt{fork}(_) \qquad e \notin \mathit{Val}}{\mathsf{wp}^{\tau}_{\mathscr{E}}\, e\, \{\Phi\}}$$

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

$$\text{MU-OM-FORK}\quad \frac{\mathsf{BOU}^{\mathscr{E}}_{\mathsf{SB}} \{P * \mathsf{ph}\, \tau\, \pi * \mathsf{obls}\, \tau\, R * \exists d.\, \mathsf{bar}\, \pi\, d\}}{\mathsf{MU}^{\tau,\tau'}_{\mathscr{E}} \{P * \exists \pi_1, \pi_2.\, \mathsf{ph}\, \tau\, \pi_1 * \mathsf{obls}\, \tau\, R \setminus R' * \mathsf{ph}\, \tau'\, \pi_2 * \mathsf{obls}\, \tau'\, R' * \pi \prec_{\mathsf{P}} \pi_1 \wedge \pi \prec_{\mathsf{P}} \pi_2\}}$$

Figure 4.17: Full form of Lawyer rules

Note that the mask $\mathscr{E}$ denotes the invariants of the *user* of a logically atomic method. In that way, when the method executes the commit viewshift, it reestablishes these user invariants. For example, when we derive the lock specification in Figure 4.15 from the logically atomic one in Figure 4.18, we maintain the invariant that the acquired lock is always eventually released. To enforce that, we need to assign a release obligation to the acquiring thread by taking OMS-CREATES-SIGNAL, storing the fresh signal identifier in the invariant. To do that upon the lock acquisition, we need to allow taking `OM` transitions at the linearization point. For that, TAU-1 allows proving the viewshift establishing $\Phi$ under the $\mathsf{BOU}^{\emptyset}_{b}\{\ldots\}$ modality, allowing taking up to $b$ small transitions before the viewshift. Note that creating an obligation requires the ownership of the current set of obligations, and some other small transitions require the ownership of phase — therefore, execution of commit requires providing $\mathsf{obls}\, \tau\, \mathscr{O} * \mathsf{ph}\, \tau\, \pi$; since these resources might be used after the linearization point, $\Phi$ is supposed to include both of them.

TAU-1 allows aborting a number of times before the linearization point, similar to a usual safety atomic update. However, as opposed to the safety setting, the number of aborts is restricted: in Lawyer, every execution step consumes fuel. In case of aborts caused by busy-waiting (like the one in ticketlock's `acquire`) the required amount of fuel is not known in advance. Therefore, we need to allow replenishing fuel until the linearization point. More generally, we need to allow taking `OM` steps upon aborts, just like we do upon the commit. To support that, we introduce a more general version of TAUs:

$$\overset{b}{\underset{\tau \mapsto \pi}{}}\left\langle \vec{x}.\, P\, [U \rightsquigarrow V] \mid \vec{y}.\, Q \Rrightarrow \Phi \right\rangle^{\mathscr{O}}_{\mathscr{E}} \tag{TAU-2}$$

The added "wait clause" $U \rightsquigarrow V$ denotes that, besides aborting or committing,

such TAU allows *waiting*. This new operation is a variation of abort that allows taking OM steps before giving $P$ back and reestablishing TAU. To wait, one proves the *wait precondition* $U$ for the current witness $\vec{x}$ and in return obtains, under the BOU modality, the *wait postcondition* $V$ for $\vec{x}$. Similarly to committing, waiting requires the ownership of phase and obligations resources.

The wait clause describes how the TAU is used before the linearization point. For example, `acquire` of ticketlock busy-waits for all previous lock owners to run `release`. Therefore, the TAU in `acquire` precondition has a wait clause that allows generating fuel by expecting on the current release obligation, *i.e.* $U \approx \mathsf{ep}\, s_{rel}\, \pi\, d, V \approx \mathsf{bar}\, \pi\, d$ for some $s_{rel}, d$; we'll define the exact clause in Section 4.7. Moreover, wait clauses allow expressing unlimited waiting ($U \triangleq \mathsf{True}, V \triangleq \mathsf{bar}\, \pi\, d$ for some $d$) and the absence of waiting ($U \triangleq \mathsf{False}$, implicit at TAU-1).

Finally, TAUs support fractional phases (used to verify ticketlock) and changing the set of thread's obligations before the linearization point (used by Sassy to verify cohortlock). As a result, we get the following definition.

**Definition 7** (Total atomic update)**.**

$$\begin{aligned}
&{}^{b}_{\tau \mapsto^{q}_{\pi}} \left\langle \vec{x}.\, P\,[U \rightsquigarrow V] \mid \vec{y}.\, Q \Rrightarrow \Phi \right\rangle^{\mathscr{O};L}_{\mathscr{E}} \triangleq \mathsf{gfp}(\mathrm{T}.\mathsf{TAU}_{\mathsf{acc}}(\mathrm{T})), \textit{where}\\
&\mathsf{TAU}_{\mathsf{acc}}(\mathrm{T}) \triangleq {}_{\mathscr{E}}\!\Vdash\!\!\Rightarrow_{\emptyset} \exists \vec{x}.\, P * (\mathsf{Abort} \wedge \mathsf{Wait} \wedge \mathsf{Commit}),\\
&\quad \mathsf{Abort} \triangleq P \;{}_{\emptyset}\!\Rrightarrow\!\!\!\!\ast_{\mathscr{E}}\; \mathrm{T},\\
&\quad \mathsf{Wait} \triangleq \forall \mathscr{O}', q' \le q.\; \mathsf{obls}\, \tau\, \mathscr{O}' * L \preceq_{\mathsf{L}} \mathsf{sgn}\, \mathscr{O}' * \mathsf{ph}_{q'}\, \tau\, \pi * U \mathrel{-\!\!*}\\
&\qquad \mathsf{BOU}^{\emptyset}_{b}\, \{V * \mathsf{ph}_{q'}\, \tau\, \pi * \mathsf{obls}\, \tau\, \mathscr{O}' * \mathsf{Abort}\}\\
&\quad \mathsf{Commit} \triangleq \forall q' \le q, \vec{y}.\; \mathsf{obls}\, \tau\, \mathscr{O} * \mathsf{ph}_{q'}\, \tau\, \pi \mathrel{-\!\!*} Q \mathrel{-\!\!*} \mathsf{BOU}^{\emptyset}_{b}\, \{{}_{\emptyset}\!\Rrightarrow_{\mathscr{E}} \mathsf{ph}_{q-q'}\, \tau\, \pi \mathrel{-\!\!*} \Phi\}
\end{aligned}$$

The following rules help to prove and use a total atomic update. From now on, TAU denotes an arbitrary (or the specific one, depending on the context) instance of Definition 7.

$$\frac{\mathsf{persistent}(C) \qquad C * R \vdash \mathsf{TAU}_{\mathsf{acc}}(R)}{C * R \vdash \mathsf{TAU}}\ \text{TAU-INTRO} \qquad\qquad \frac{\mathsf{TAU}_{\mathsf{acc}}(\mathsf{TAU})}{\mathsf{TAU}}\ \text{TAU-ELIM}$$

### Logically Atomic Specification for Fair Lock

We define the logically atomic version of fair locks in Figure 4.18. Note that, to avoid introducing additional notation, there we inline the definition of total correctness logically-atomic triples (TCLATs), presenting the specifications acquireSpecTAU and releaseSpecTAU as usual Hoare triples parameterized by TAUs. The full definition of corresponding TCLATs is inspired by that of [28] and can be found in the accompanying Rocq development.

Similarly to Figure 4.15, the atomic lock specification in Figure 4.18 has a number of parameters. Among them is a set of obligations levels $L_{\mathtt{acq}}$ that `acquire` might

rely on internally, "module degree" $d_{\mathtt{FL}}$, and a "module bound function" $\mathscr{B}_{\mathtt{FL}}$. Besides these, $d_r$ and $d_w$ degrees are used alongside the rounds mechanism described below.

The main difference with Figure 4.15 is the resource $\mathsf{FL}(r,b)$ representing the abstract state $(r,b)$ of the lock: the first component denotes the current "round" (described below), and the second denotes whether the lock is currently held. The logical atomicity of lock is specified wrt. this resource: changes to this resource correspond to linearization points of `acquire` and `release`, upon which the lock client can update its invariants (which in particular allows deriving the specification in Figure 4.15). While `acquire` waits for the lock, both the round and the state of the lock might change (due to concurrent interference), and upon the acquisition (linearization point) the state changes from $\mathsf{False}$ to $\mathsf{True}$ with round remaining intact. This is specified by the $\left\langle (r,b).\, \triangleright \mathsf{FL}(r,b)\,[\ldots] \mid \mathsf{FL}(r,\mathsf{True}) * \neg b \Rrightarrow \ldots \right\rangle$ part of $\mathsf{TAU}_{\mathsf{acq}}$. Then, when the lock is released in `release`, lock state changes from $\mathsf{True}$ to $\mathsf{False}$, and the round increases, as can be seen in $\mathsf{TAU}_{\mathsf{rel}}$.

To understand the purpose of rounds, consider the "round wait clause" $\mathsf{RWC}(RR)$ parameterized by user-provided "round resource" $RR : \mathbb{N} \to \mathit{iProp}$ and used in `acquire` specification. Its purpose is to allow generating fuel of "waiting degree" $d_w$ to justify the waiting that `acquire` performs while $b = \mathsf{True}$. To use it, the module must provide the wait precondition of $\mathsf{RWC}(RR)$, which can be done in two ways. First, the module can spend a barrel of "reserved degree" $d_r$ — with that, it obtains the round resource for current $r$ and a barrel of $d_w$. It is up to the client to provide an $RR$ that allows such an operation; in practice, $RR$ consists of an expectation permission and some client-specific resource. Second, the module can reuse the previously established round resource and generate fuel for free (*e.g.*by expecting on an already existing expectation permission). The rounds change every time the lock is released and therefore require to keep spending $d_r$ fuel — however, the feature of *fair* locks is that they can determine the number of rounds before the given thread acquires the lock and can therefore reserve the needed amount of $d_r$ fuel.

### Verification of lock client program

Here we explain the most important steps of verifying `client` in Figure 4.13 on top of an abstract fair lock specification (Figure 4.15). Recall that we assume the following holds for the lock and `OM` parameters:

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

With that, we give `client` the following specification:

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

$$\forall \tau, \pi, u. \quad \begin{array}{l} \{\mathsf{bar}\,\pi\, d_{\mathtt{FL}} * \mathsf{ph}\,\tau\,\pi * \mathscr{B}_{\mathtt{FL}}(u) \le \mathsf{SB}\} \\ \quad \texttt{newLock}\ () \\ \left\{ lk.\ \mathsf{TAULock}^{lk}(u) * \mathsf{FL}(0, \mathsf{False}) * \mathsf{ph}\,\tau\,\pi \right\}^{\tau} \end{array} \qquad \text{(newlockSpecTAU)}$$

where

$$\mathsf{TAULock}^{lk}(u) \triangleq \exists \mathsf{locked} : \mathit{iProp},\ \iota_{\mathtt{FL}} : \mathscr{N},\ \mathsf{FL} : \mathbb{N} \to \mathbb{B} \to \mathit{iProp}.$$

$$\left( \begin{array}{l} \forall \tau, \mathscr{O}, \pi, RR, \Phi. \\ \left\{ \mathsf{obls}\,\tau\,\mathscr{O} * \mathsf{ph}\,\tau\,\pi * \mathsf{bar}\,\pi\, d_{\mathtt{FL}} * L_{\mathsf{acq}} \prec_{\mathsf{L}} \mathsf{sgn}\,\mathscr{O} * \mathsf{TAU}_{\mathsf{acq}} \right\} \texttt{acquire}\ lk\ \{\Phi\}^{\tau} \\ \quad \text{where} \\ \mathsf{TAU}_{\mathsf{acq}} \triangleq {}^{u}_{\tau \mapsto \pi}\left\langle (r,b).\ \triangleright \mathsf{FL}(r,b)\,[\mathsf{RWC}(RR)] \mid \mathsf{FL}(r, \mathsf{True}) * \neg b \Rrightarrow \mathsf{locked} \mathrel{-\!\!*} \Phi \right\rangle^{\mathscr{O}; L_{\mathsf{acq}}}_{\top \setminus \iota_{\mathtt{FL}}} \\ \mathsf{RWC}(RR) \triangleq b = \mathsf{True} * (\mathsf{bar}\,\pi\, d_r \vee RR(r)) \rightsquigarrow \mathsf{bar}\,\pi\, d_w * RR(r) \end{array} \right) \wedge \qquad \text{(acquireSpecTAU)}$$

$$\left( \begin{array}{l} \forall \tau, \mathscr{O}, \pi, \Phi. \\ \{\mathsf{obls}\,\tau\,\mathscr{O} * \mathsf{ph}\,\tau\,\pi * \mathsf{bar}\,\pi\, d_{\mathtt{FL}} * \mathsf{locked} * \mathsf{TAU}_{\mathsf{rel}}\} \texttt{release}\ lk\ \{\Phi\}^{\tau} \\ \quad \text{where} \\ \mathsf{TAU}_{\mathsf{rel}} \triangleq {}^{u}_{\tau \mapsto \pi}\left\langle r.\ \triangleright \mathsf{FL}(r, \mathsf{True}) \mid \mathsf{FL}(r+1, \mathsf{False}) \Rrightarrow \Phi \right\rangle^{\mathscr{O}}_{\top \setminus \iota_{\mathtt{FL}}} \end{array} \right) \qquad \text{(releaseSpecTAU)}$$

Figure 4.18: TAU-based fair lock specification (parameterized by $(d_w \prec_{\mathsf{D}} d_r \prec_{\mathsf{D}} d_{\mathtt{FL}} : \mathsf{Degree})$, $L_{\mathsf{acq}} : \mathit{FinSet}(\mathsf{Level})$, $\mathscr{B}_{\mathtt{FL}} : \mathbb{N} \to \mathbb{N}$)

```
1  {obls τ ∅ ∗ ph τ π ∗ 3·bar π d_CL}
2  let client () =
3    let flag = ref false in let lk = newLock () in
4    {□ isLock(4, ∃b. ClRes(b)) ∗ □ sgn_∃ s_f l_f ∗ obls τ {s_f} ∗ ph τ π ∗ 4·bar π d_FL ∗ 2·bar π d_CL}
5    (∗ for some τ' and π_1 ≻_P π, π_2 ≻_P π: ∗)
6    fork ({obls τ' ∅ ∗ ph τ' π_2 ∗ 2·bar π_2 d_CL} right flag lk {obls τ' ∅ ∗ ph τ' π_2});
7    {obls τ {s_f} ∗ ph τ π_1 ∗ 3·bar π_1 d_FL} left flag lk {obls τ ∅ ∗ ph τ π_1}
8  {∃π' ≻_P π. obls τ ∅ ∗ ph τ π'}
```

Figure 4.19: Sketch of client program verification

To verify this specification, we will use the proposition $\exists b.\ \mathsf{ClRes}^{\mathit{flag}, s_f}(b)$ as the resources protected by the lock, where $\mathsf{ClRes}^{\mathit{flag}, s_f}(b : \mathbb{B}) \triangleq \mathit{flag} \mapsto b * \mathsf{sgn}\, s_f\, l_f\, b$. To reduce clutter, we omit the flag and signal parameters of ClRes once they are established.

The annotated program in Figure 4.19 sketches the high-level proof. In the rest of the section, we emphasize the use of lock specifications; for the complete proof,

we refer the reader to [95]. Initially, after allocating the location `flag`, we create the obligation $s_f$ which allows us to prove $\mathsf{ClRes}^{flag,s_f}(\mathsf{False})$. Moreover, we initialize the lock using (*newlockSpec*), choosing $u \triangleq 4, P \triangleq \exists b.\, \mathsf{ClRes}^{flag,s_f}(b)$. To obtain fuel for all these steps, we exchange one of the initial barrels of $d_{\mathtt{CL}}$.

To prove the left thread's specification (line 7 in Figure 4.19), we prove the following triples (additionally assuming persistent $\mathsf{isLock}(4, \exists b.\, \mathsf{ClRes}(b)) * \mathsf{sgn}_\exists\, s_f\, l_f$ and generalizing over the phase):

$$\left\{\mathsf{obls}\,\tau\left\{s_f\right\} * \mathsf{ph}\,\tau\,\pi * \mathsf{bar}\,\pi\,d_{\mathrm{FL}}\right\}\ \mathtt{acquire}\ lk\ \left\{\exists s_r.\ \begin{array}{c}(\exists b.\, \mathsf{ClRes}(b)) * \mathsf{locked}(s_r) *\\ \mathsf{obls}\,\tau\left\{s_f, s_r\right\} * \mathsf{ph}\,\tau\,\pi\end{array}\right\}^{\tau}$$

$$\left\{\begin{array}{c}\mathsf{ClRes}(b:\mathbb{B}) * \mathsf{bar}\,\pi\,d_{\mathrm{FL}} *\\ \mathsf{obls}\,\tau\left\{s_f, s_r\right\} * \mathsf{ph}\,\tau\,\pi\end{array}\right\}\ \mathtt{flag\ :=\ true}\ \left\{\mathsf{ClRes}(\mathsf{True}) * \mathsf{obls}\,\tau\left\{s_r\right\} * \mathsf{ph}\,\tau\,\pi\right\}^{\tau}$$

$$\left\{\begin{array}{c}\mathsf{ClRes}(\mathsf{True}) * \mathsf{bar}\,\pi\,d_{\mathrm{FL}} *\\ \mathsf{obls}\,\tau\left\{s_r\right\} * \mathsf{ph}\,\tau\,\pi * \mathsf{locked}(s_r)\end{array}\right\}\ \mathtt{release}\ lk\ \left\{\mathsf{obls}\,\tau\,\emptyset * \mathsf{ph}\,\tau\,\pi\right\}^{\tau}$$

The first one is shown by applying *acquireSpec*; the premise $L_{\mathrm{FL}} \prec_{\mathsf{L}} \mathsf{sgn}\left\{s_f\right\}$ is satisfied by $\mathsf{sgn}_\exists\, s_f\, l_f$ and the $\forall l \in L_{\mathrm{FL}}.\ l \prec_{\mathsf{L}} l_f$ assumption. To prove the second one, we unfold ClRes to get the points-to and signal state resources, set the flag and fulfill the obligation along with setting the signal. Finally, we prove the third one by applying *releaseSpec*. For that, we prove, for any $q$, the following BOU simply by using BOU-INTRO:

$$\frac{\mathsf{ClRes}(\mathsf{True}) * \mathsf{ph}_q\,\tau\,\pi * \mathsf{obls}\,\tau\,\emptyset}{\mathsf{BOU}_4^{\top\setminus\{\iota_{\mathrm{FL}}\}}\left\{\mathsf{ClRes}(\mathsf{True}) * (\mathsf{ph}_{1-q}\,\tau\,\pi \mathrel{-\!\!*} \mathsf{obls}\,\tau\,\emptyset * \mathsf{ph}\,\tau\,\pi)\right\}}$$

As Figure 4.13 shows, the right thread allocates a local variable $c$ and repeatedly stores in it, within `wait`, the current value of the flag, until the flag is set. The key to proving the right thread's specification (line 6 in Figure 4.19) is verifying an iteration of `wait`:

$$\left\{\begin{array}{c}\mathsf{isLock}(4, \exists b.\, \mathsf{ClRes}(b)) * \mathsf{obls}\,\tau\,\emptyset * \mathsf{ph}\,\tau\,\pi * 3{\cdot}\mathsf{bar}\,\pi\,d_{\mathrm{FL}} *\\ \mathsf{sgn}_\exists\, s_f\, l_f * \mathsf{ep}\, s_f\,\pi\,d_{\mathrm{FL}} * c \mapsto \mathsf{False}\end{array}\right\}$$
$$\mathtt{acquire}\ lk;\ c := !flag;\ \mathtt{release}\ lk$$
$$\left\{\mathsf{obls}\,\tau\,\emptyset * \mathsf{ph}\,\tau\,\pi * \exists b.\, c \mapsto b * (b = \mathsf{False} \Rightarrow 4{\cdot}\mathsf{bar}\,\pi\,d_{\mathrm{FL}})\right\}^{\tau}$$

We can derive it by proving three triples (assuming $\mathsf{isLock}(4, \exists b.\, \mathsf{ClRes}(b)) * \mathsf{sgn}_\exists\, s_f\, l_f *$ $\mathsf{ep}\, s_f\,\pi\,d_{\mathrm{FL}}$ and generalizing over the phase and thread):

$$\left\{\mathsf{obls}\,\tau\,\emptyset * \mathsf{ph}\,\tau\,\pi * \mathsf{bar}\,\pi\,d_{\mathrm{FL}}\right\}\ \mathtt{acquire}\ lk\ \left\{\exists s_r.\ \begin{array}{c}(\exists b.\, \mathsf{ClRes}(b)) * \mathsf{obls}\,\tau\left\{s_r\right\} *\\ \mathsf{locked}(s_r) * \mathsf{ph}\,\tau\,\pi\end{array}\right\}^{\tau}$$

$$\left\{\begin{array}{c}c \mapsto \mathsf{False} * \mathsf{ClRes}(b) *\\ \mathsf{bar}\,\pi\,d_{\mathrm{FL}} * \mathsf{ph}\,\tau\,\pi\end{array}\right\}\ \mathtt{c\ :=\ !flag}\ \left\{c \mapsto b * \mathsf{ClRes}(b) * \mathsf{ph}\,\tau\,\pi\right\}^{\tau}$$

$$\left\{\begin{array}{c}\mathsf{ClRes}(b) * \mathsf{locked}(s_r) * \mathsf{bar}\,\pi\,d_{\mathrm{FL}} *\\ \mathsf{ph}\,\tau\,\pi * \mathsf{obls}\,\tau\left\{s_r\right\}\end{array}\right\}\ \mathtt{release}\ lk\ \left\{\begin{array}{c}\mathsf{obls}\,\tau\,\emptyset * \mathsf{ph}\,\tau\,\pi *\\ (\neg b \Rightarrow 4{\cdot}\mathsf{bar}\,\pi\,d_{\mathrm{FL}})\end{array}\right\}^{\tau}$$

The first one is proven by applying *acquireSpec*; its $L_{\mathrm{FL}} \prec_{\mathsf{L}} \mathsf{sgn}\,\emptyset$ premise holds trivially. To prove the second one, we unfold $\mathsf{ClRes}$ to read the flag and store its value in $c$; to get fuel for these two steps, we exchange a barrel of $d_{\mathrm{FL}}$ into two barrels of $d_0$. The last one is proven by applying *releaseSpec*. Most of its premise is satisfied immediately; we are left proving, for some $q$,

$$\frac{\mathsf{ph}_q\,\tau\,\pi * \mathsf{obls}\,\tau\,\emptyset * \mathsf{ClRes}(b)}{\mathsf{BOU}_4^{\top\setminus\{\iota_{\mathrm{FL}}\}}\{\mathsf{ClRes}(b) * (\mathsf{ph}_{1-q}\,\tau\,\pi \mathrel{-\!\!*} \mathsf{obls}\,\tau\,\emptyset * \mathsf{ph}\,\tau\,\pi * (b = \mathsf{False} \Rightarrow 4{\cdot}\mathsf{bar}\,\pi\,d_{\mathrm{FL}}))\}}$$

which, after framing a fragment of phase and the points-to part of $\mathsf{ClRes}$, becomes

$$\frac{\mathsf{ph}_q\,\tau\,\pi * \mathsf{obls}\,\tau\,\emptyset * \mathsf{sgn}\,s_f\,l_f\,b}{\mathsf{BOU}_4^{\top\setminus\{\iota_{\mathrm{FL}}\}}\{\mathsf{sgn}\,s_f\,l_f\,b * \mathsf{obls}\,\tau\,\emptyset * \mathsf{ph}_q\,\tau\,\pi * (b = \mathsf{False} \Rightarrow 4{\cdot}\mathsf{bar}\,\pi\,d_{\mathrm{FL}})\}}$$

Here, the proof proceeds similarly to that of `incr_loop` from Section 4.3. If $b = \mathsf{True}$, we simply apply BOU-INTRO. If $b = \mathsf{False}$, we apply BOU-OU followed by OU-EXPECTS-EP, repeating it 4 times, thus obtaining the needed amount of fuel. Note that we can only prove this $\mathsf{BOU}$ *after* fulfilling the obligation (to satisfy the obligations levels restriction of OU-EXPECTS-EP), but *before* returning $\mathsf{ClRes}$ (that holds the signal state resource) — exactly what *releaseSpec* allows.

# Chapter 5

# Verifying Wait-Freedom for Concurrent Higher-Order Programs

**Abstract**

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

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

We demonstrate the effectiveness of our approach by proving wait-freedom for several representative examples, including a higher-order list map function, and a memory-efficient single-producer, single-consumer queue. For the latter, wait-freedom is conditional in that, as the name suggests, there can be at most one enqueuer thread and one dequeuer thread. To capture this formally, we introduce the notion of restricted wait-freedom as a variant of wait-freedom

that restricts the number of concurrent threads, and show how our approach can support reasoning about restricted wait-freedom. All our results have been mechanized on top of the Rocq Prover and using the Iris separation logic framework that Lawyer is also based on.

## 5.1 Introduction

Any implementation of a concurrent data structure must account for possible interference between multiple operations on that data structure running concurrently. Done wrongly, it might lead to incorrect behavior of operations, or even worse, to undefined behavior of an entire program. The simplest way to ensure concurrent operations' correctness is to resort to *blocking concurrency*, *e.g.*, using a lock, allowing only one operation to progress at any given moment, and forcing others to wait for its completion. However, in many settings, *e.g.*, in operating system kernels [14], blocking concurrency is discouraged as it might lead to poor performance, and due to the risk of deadlocking the entire system. Therefore, in such cases a *lock-free* implementation is more desirable. Lock-free data structure implementations guarantee that among the concurrently running operations, at least one of them must run to completion. Yet, lock-freedom does not guarantee that there is no starvation. That is, a specific call to a lock-free data structure's operation might not terminate because it is starved by other calls. To avoid starvation, one must use so-called *wait-free* [45] implementations. Wait-freedom is the strongest form of non-blocking concurrency. It guarantees that *every* individual call to the data structure operations must run to completion regardless of how many other operations of the same data structure run concurrently with it.

We illustrate the differences between blocking, lock-free, and wait-free concurrency using a simple and contrived example: a concurrent counter implementation featuring an increment operation (and a read operation which we elide). Figure 5.1 shows multiple possible implementations of the increment operation, all of which store the current value of the counter under the location $l$. The implementation in Figure 5.1a allows only one thread at a time to access the counter, forcing all others to wait until the lock is released. The lock-free implementation (Figure 5.1b), implemented via an atomic Compare-And-Set (CAS) operation in a loop, does not wait for other concurrent operations on the counter to complete. It attempts to increment using the CAS operation, and if that fails, it retries. This implementation is lock-free: if the CAS operation fails, then the counter value must have changed after it has been read on the line above the CAS operation, which in turn means that another increment operation must have succeeded. However, this lock-free implementation can starve threads, even under some fair schedulers. (Even though unlikely, a pathological but fair scheduler could schedule the thread, and subsequently preempt it right before the CAS operation, only to return to it, every time, after another increment has succeeded.) Finally, we have the wait-free implementation in Figure 5.1c. It uses the atomic Fetch-And-Add (FAA) operation which atomically increments the counter. Thus, the increment always finishes in a single, atomic step. In our discussions throughout the rest of the paper we will refer to this wait-free implementation of increment. To this end, we introduce

```
let incr_block l =
  acquire m;
  let n = !l in
  l := n + 1;
  release m;
  n
```

(a) Blocking implementation. `m` is a lock instance

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

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

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

let incr l () =
  incr_waitfree l
```

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

Figure 5.1: Comparison of counter increment implementations; we use an OCaml-like syntax. Location `l` stores an integer

a wrapper function `incr`, so that we can treat `incr`$l$ as a function (that ignores its argument and executes increment).

In this contrived example of a simple concurrent counter (Figure 5.1c) we essentially relegate the problem of making the data structure wait-free to the hardware using the FAA operation. In reality, as we will see below, for data structures ever so slightly more complex than this, wait-free implementations are far from trivial to implement and to formally reason about. Indeed, such implementations must be designed to execute concurrently in a way that neither do they have to wait for the completion of other operations, nor are they impeded in a way by the other operations that force them to retry. Therefore, a wait-free implementation of a data structure is often much more complicated than a lock-free or blocking implementation of the same data structure (see *e.g.* wait-free queue of Kogan and Petrank [69] versus the lock-free queue of Michael and Scott [89]). Thus, it is often also more challenging to verify.

In this paper, we show how to use a higher-order concurrent program logic to verify wait-freedom for programs written in a concurrent *higher-order* programming language with general references. Such languages achieve greater modularity by supporting features such as closures and first-class modules. Our work goes significantly beyond the state of the art [80, 101] which has only considered wait-freedom of programs in first-order programming languages without general references. Working with higher-order programs, and with general references, introduces many new challenges that one needs to answer. In fact, it is not even clear a priori how to formally define wait-freedom in our setting. Informally, it is often explained that a wait-free operation should be able to make progress in any program context under any scheduler. However, it is not entirely clear here what one means by "any program context", in particular in our higher-order setting. A "context" (or *client*) of an operation can be any program that calls that operation, either directly or indirectly (through a reference), possibly concurrently—see *e.g.*, the `good_client` in Figure 5.2. However, it does not make sense to consider literally any client. In particular, the client's behavior should be "safe" in that it should not be able to violate data encapsulation of the wait-free data structure, *i.e.*, it should not be able to directly access the data structure's internal

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

let good_client =
  let r = ref simple_op in
  simple_op () || !r ()

let bad_client =
  [0x42] := false || simple_op ()
```

Figure 5.2: An example of a wait-free operation together with two possible clients. Above, || is a parallel composition and $[\cdot]$ is a direct memory access.

state. For a simple example of what can go wrong see the `bad_client` in Figure 5.2 which directly writes to `0x42` while concurrently running `simple_op`. The example `bad_client` could potentially violate an invariant of `simple_op` (namely the invariant that the local state variable `b` is always true) and thus break the wait-freedom of `simple_op`.

The example `bad_client` in Figure 5.2 shows that the progress guarantee of a wait-free operation crucially relies on its client being safe. Indeed, prior works on wait-freedom verification for first-order programs enforce safety of clients in one way or another, but their approaches do not apply in our setting. One way to enforce clients' safety is by using a programming language that syntactically separates accesses to the operation's and client's states [80]. This approach, however, is not viable for the higher-order language we consider. (In general, the introduction of higher-order store enables behaviors not present in a first-order setting, including turning a terminating program into non-terminating, as demonstrated by [27]). Another approach to ensuring safety of clients is by requiring them to be verified [99, 101]. In addition to not being applicable in our higher-order setting, this approach also inherently limits the applicability of the wait-freedom result: now every single client of potential interest must be verified in the corresponding framework. Rather, we take an approach that does not require one to verify individual clients but instead establishes operations' progress within arbitrary clients. This also includes those clients out of reach of existing solutions, *e.g.*, higher-order clients.

In this work, we present the first solution to the verification of wait-freedom for a realistic, higher-order language. Our solution is based on the recent Lawyer logic [94], an extension of the Trillium [118] and Iris [62] frameworks. So far, Lawyer has only been used for establishing (fair) termination of closed programs. The key insight of our work is that a specific form of specifications given to a function forces that function to terminate *in any context, without either blocking or starving*. That is, such Lawyer specifications essentially codify, for all intents and purposes, wait-freedom in the Lawyer program logic. However, there are two main technical issues that prevent the Lawyer logic from directly being applicable to proving wait-freedom. Firstly, the so-called adequacy theorem of Lawyer (that reduces establishing termination to proving a specification) is not applicable. This theorem applies to a closed program, *i.e.*, not a

standalone data structure, but rather the data structure and its client. Moreover, the adequacy theorem states that when a program is proven correct in Lawyer, then all of its threads must terminate, which does not consider non-terminating clients. Hence, in this work we state and prove a separate and novel adequacy theorem that establishes wait-freedom of a standalone operation. Secondly, the wait-freedom result should apply to an arbitrary, unknown client of the operation — but, as we discussed above, it only makes sense when the client respects state encapsulation. Therefore, in our approach to verification of wait-freedom we prove once and for all that *all programs in our programming language respect state encapsulation* (if they do not use pointer arithmetic). For this purpose we construct a so-called logical relations model for our programming language. This is similar to how prior works on proving robust safety of programs [34, 116] have used logical relations models.

We use the approach outlined above to prove wait-freedom for a number of example programs written in our higher-order language, including those that highlight these higher-order programming features. In particular, we modularly verify wait-freedom of the higher order map function for lists. That is, we show that `list_mapf` is wait-free whenever it is applied to any wait-free argument `f`. We also show that our approach supports practical wait-free algorithms by verifying a memory-efficient single-producer, single-consumer queue [57]. An important point about this example, and in fact many practical wait-free algorithms, is that these algorithms in fact assume fixed upper bounds on the number of concurrent operations. Verification of this class of wait-free algorithms has been largely ignored by the prior work on verification of wait-freedom. Supporting formal reasoning about such examples requires us to introduce a notion of what we call *restricted* wait-freedom, and to slightly adjust our approach.

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

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

The rest of the paper is organized as follows. We formally define wait-freedom in Section 5.2 and also state the adequacy theorem that reduces proving wait-freedom of an operation to proving its wait-freedom specification in Lawyer. Then, in Section 5.3 we provide the necessary background on Iris. We then illustrate our approach in

Section 5.4 by proving the wait-freedom specification for a simple example program. In Section 5.5 we continue by verifying more complicated programs which require adapting both the definition of wait-freedom and the Lawyer specification as we explained earlier. We explain the key ideas for proving our adequacy theorem in Section 5.6. Finally, we discuss the limitations of our solution in Section 5.7, compare it with related work in Section 5.8, and conclude in Section 5.9.

## 5.2 Overview

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

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

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

### Definition of Wait-Freedom for Our Language

We consider the programs written in a higher-order ML-like language $\lambda_{\mathsf{ref}}^{\mathsf{conc}}$. An excerpt of its semantics is shown in Figure 5.3, for the full semantics see our Rocq development. Among the set *Expr* of $\lambda_{\mathsf{ref}}^{\mathsf{conc}}$ expressions, there are primitives for working with heap (*e.g.* $!e$ and **`free`**$\,e$) and for concurrency (**`fork`**$\,(e)$; the parallel composition $||$ used in the examples in Section 5.1 is a derived construction [12]). A less common feature of $\lambda_{\mathsf{ref}}^{\mathsf{conc}}$ — **`to_int`** primitive — is explained further in Section 5.5. The heap is modeled as a partial map from locations to language values *Val* (which might include closures). In turn, the thread pool is modeled as a partial map from thread identifiers Thread to expressions; the per-thread reductions are naturally lifted to configurations *Config* (thread pools together with heaps). The structure of evaluation contexts *ECtx* establishes right-to-left call-by-value evaluation order. The $\mathit{fill}\,(E : \mathit{ECtx})\,(e : \mathit{Expr})$ operator fills the hole $\bullet$ in the context $E$ with the expression $e$;

**Syntax**

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

**Pure reductions** **Per-thread reductions**

$$
(\texttt{rec}\, f\, (x) := e)\, v \overset{\text{pure}}{\rightsquigarrow} e[\texttt{rec}\, f\, (x) := e/f][v/x] \qquad (e, h) \rightsquigarrow (e', h) \;\; (\text{if } e \overset{\text{pure}}{\rightsquigarrow} e')
$$

$$
\texttt{fst}(v_1, v_2) \overset{\text{pure}}{\rightsquigarrow} v_1 \qquad (!\,\ell, h[\ell \mapsto v]) \rightsquigarrow (v, h[\ell \mapsto v])
$$

$$
\vdots \qquad\qquad \vdots
$$

**Configuration reductions**

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

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

Figure 5.3: An excerpt of $\lambda_{\mathsf{ref}}^{\mathsf{conc}}$ semantics

we avoid the more common $E[e]$ notation, as in this paper it is used for lookups (see above).

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

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

**Definition 8** (Call and return in a thread pool)**.**

$$Call(\mathscr{E}, \tau, E, \mathtt{op}) \triangleq \exists a \in Val.\ \mathscr{E}[\tau] = fill\, E\, (\mathtt{op}\, a)$$
$$Ret(\mathscr{E}, \tau, E) \triangleq \exists r \in Val.\ \mathscr{E}[\tau] = fill\, E\, r$$

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

**Definition 9** (Eventual return of calls)**.** *For an operation* op :*Val, execution trace etr and thread identifier* $\tau$*, we say that all* $\tau$*'s calls to* op *in etr eventually return iff*

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

Above, $schedUntilRet(etr, \tau, E, i)$ says that, if the thread $\tau$ has called (in terms of Definition 8) op at index $i$ and has not returned from it yet, then $\tau$ is guaranteed to be eventually scheduled. Note that each call requires a separate return, matched by an evaluation context. Indeed, in $\lambda_{\mathsf{ref}}^{\mathsf{conc}}$, an evaluation context is preserved by configuration reductions until the expression under that context reduces to a value. Therefore, the contexts of, *e.g.*, nested calls will be different (with every subsequent call occurring in a "deeper" context).

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

- The program of every thread must be a substitution of the wait-free operation's code into some expression with a free variable $x$. Moreover, this expression must not contain hard-coded locations and pointer arithmetic (denoted by *noLocs*): the example from Figure 5.2 shows why it is necessary to restrict random memory access in clients.
- The data structure accessed by the operation must be initialized. What exactly "initialization" means depends on the operation (see an example below), therefore our definition of wait-freedom is parameterized with the set of initialized configurations.

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

**Definition 10** (Wait-freedom)**.** *Given a set of initialized configurations* $C \subseteq Config$*, we define wait-freedom of an operation* op *as*

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

For example, in Section 5.4 we show that a FAA-based `incr` operation of counter (Figure 5.1c) is wait-free if the counter location $l$ is initialized with an integer. In terms of Definition 10, it is denoted as $\mathit{waitFree}_{\mathit{hasInt}(l)}(\texttt{incr}\ l)$, where $\mathit{hasInt}(l) \triangleq \{c \mid c.2[l] \in \mathbb{Z}\}$.

Note that, while Definition 10, inspired by the classical definition by Herlihy [45], captures the behavior of simple wait-free operations, more complicated classes of algorithms only satisfy weaker forms of that property. We will see examples of it in Section 5.5.

### Proving Wait-Freedom with Lawyer Logic

Definition 10 captures wait-freedom at the level of execution traces, which are notoriously hard to reason about directly. To avoid that, our approach reduces establishing wait-freedom of a given operation to verifying this operation with Lawyer [94] — a concurrent separation logic built on top of Iris [62] and Trillium [118] frameworks. We explain the details of Lawyer specifications and logic in the subsequent sections; here, we present the high-level form of the specification that a wait-free operation must satisfy.

**Definition 11** (Lawyer specification of wait-freedom)**.** *For a set of initialized configurations $C \subseteq \mathit{Config}$, we define the Lawyer specification of* `op`*'s wait-freedom as*

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

The above specification can be read as follows. For any initialized configuration $c$, we assume the ownership of $c$'s entire heap, represented as an Iris proposition $\mathsf{heap_repr}(c)$. Given that (reading $\Rrightarrow$ as an implication), we prove two specifications for the operation `op`. First, $\mathsf{NoInfExec}(\texttt{op})$ states that for any argument $a : \mathit{Val}$, execution of $\texttt{op}\,a$ cannot be infinite. Second, $\mathsf{PresInv}(\texttt{op})$ states that for any $a$, execution of $\texttt{op}\,a$ preserves the internal invariants that `op` relies on (*e.g.* that the location $l$ of counter in Figure 5.1c always stores a number).

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

**Theorem 4** (Adequacy)**.** *If* $\mathsf{WaitFreeSpec}_C(\texttt{op})$ *is provable in Lawyer, then* $\mathit{waitFree}_C(\texttt{op})$ *holds in the meta-logic.*

As we note in Section 5.2, the notion of wait-freedom for more complicated classes of operations is more elaborate. For these, we use slightly different versions of Definition 11 and Theorem 4 described in Section 5.5.

## 5.3 Background: Iris Logic

The Lawyer logic [94] that we use to verify wait-freedom is built on top of Trillium framework [118], which is in turn an extension of the Iris framework [62]. In this

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

Figure 5.4: Propositions of Iris logic. Here, *Expr*, *Loc* and *Val* refer to those from Figure 5.3

section, we provide the general background on Iris by proving two specifications (not relevant to wait-freedom) for the counter increment operation `incr` from Figure 5.1c. Later in Section 5.4, we actually verify the wait-freedom specification for `incr` with Lawyer logic, concentrating on how it differs from Iris. We only address Trillium framework specifically in Theorem 6.

Strictly speaking, Iris is a *framework* that only provides the minimal base logic and allows one to define custom program logics on top of it. We consider the specific instantiation of Iris (inspired by one used in Iris Lecture Notes [12]) shown in Figure 5.4, the constructs of which we explain as we go. In this section, by "Iris" we mean that specific instantiation. Similarly, there are many possible specifications for `incr` even within that Iris instantiation; we consider those that illustrate the concepts used in this paper.

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

With the intuition above, we can specify the program in Figure 5.1c as follows:

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

This specification states that regardless of the location and the current value of the counter, the increment operation executes without getting stuck, that its return value equals the current counter value, and that upon return this value is actually incremented. Indeed, the adequacy theorem of Iris [63] allows us to establish this property.

Proving simple specifications such as incr-spec-seq amounts to applying primitive proof rules on every execution step. Figure 5.5 shows a subset of Iris proof rules required to verify the specifications in this section. In particular, as execution of $\texttt{incr}\,\ell()$ consists of two beta-reductions and one FAA step, we need the rules IRIS-BETA and IRIS-FAA. The former states that taking a beta-reduction does not change the program state, so the reduct can be verified with the same pre- and post-condition.

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

$$\text{WFREE:IRIS-HT-FRAME}\ \frac{\{P\}\ e\ \{Q\}}{\{P * R\}\ e\ \{Q * R\}} \qquad \text{WFREE:IRIS-HT-CSQ}\ \frac{\{P'\}\ e\ \{Q'\} \quad P \Rightarrow P' \quad \forall v.\, Q'(v) \Rightarrow Q(v)}{\{P\}\ e\ \{Q\}}$$

$$\text{IRIS-INV-ACC}\ \frac{\{P * Q\}\ e\ \{P * R\} \quad e \text{ is atomic}}{\{\boxed{P} * Q\}\ e\ \{R\}}$$

Figure 5.5: Selected rules of Iris program logic

The latter states that fetch-and-add updates the value stored under the corresponding location and returns the previously stored value.

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

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

Below, we illustrate the use of Iris as a *concurrent separation logic* by proving a more elaborate specification for `incr`. There are many possible concurrent specifications for this operation; Iris Lecture Notes [12] provide a number of them. We choose another, *token-based* specification because the notion of "tokens" is used a few times throughout the paper: it resembles the Lawyer's notion of fuel (Section 5.4), and is used to specify restricted wait-freedom (Section 5.5). As said above, in the concurrent setting it is impossible to always hold the exact knowledge of the current counter value. However, by limiting the number of possible increments, one can approximate the current counter value when reading it. Namely, when instantiating the counter, the user chooses its maximal possible value $N$. Upon the instantiation, the user receives $N$ non-replenishable *tokens*. A token is spent on every counter increment. Unused tokens can be presented upon reading the counter; the more tokens are presented, the smaller the counter value must be. Formally, we can specify the counter operations as

follows (where $\texttt{mk_cnt}\,() \triangleq \texttt{ref}\,0$ and $\texttt{read}\,\ell \triangleq !\ell$):

$$\forall N : \mathbb{N}. \quad \{\mathsf{True}\}\ \texttt{mk_cnt}\,()\ \{\ell.\, \boxed{\mathsf{cntInvUpto}(\ell, N)} * \mathsf{toks}_N(N)\}$$
$$\forall \ell, N. \quad \{\boxed{\mathsf{cntInvUpto}(\ell, N)} * \mathsf{toks}_N(1)\}\ \texttt{incr}\,\ell\,()\ \{m.\, 0 \le m\}$$
$$\forall \ell, N, k. \quad \{\boxed{\mathsf{cntInvUpto}(\ell, N)} * \mathsf{toks}_N(k)\}\ \texttt{read}\,\ell\ \{m.\, 0 \le m \le (N-k) * \mathsf{toks}_N(k)\}$$

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

Then, $\mathsf{toks}_N(k)$ is an example of a *ghost resource*, *i.e.* one that is not directly tied to the program state. The Iris user can choose a number of ghost resources to support the verification process. In Figure 5.4, we assume that Iris instantiation contains resources for tracking tokens, where a proposition $\mathsf{toks}_N(k)$ asserts ownership of $k$ out of $N$ tokens and satisfies the following laws:

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

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

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

We omit the verification of `mk_cnt` and `read` as it follows standard principles of Iris safety proofs [12]. Below, we show how the verification of `incr` changes with the new specification. We verify the two beta-reductions as before by applying IRIS-BETA twice and end up with the following triple to prove:

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

To proceed, we need the points-to resource in the invariant. For this, we use the invariant access rule IRIS-INV-ACC that allows opening the invariant (*i.e.* obtain the underlying resource) for a single execution step and requires closing it back after that

step[1]. Applying this rule to the triple above and unfolding our invariant, we are left with proving

$$\left\{\begin{array}{c}(\exists k \geq 0.\, \ell \mapsto k * \mathsf{toks}_N(k)) \\ * \mathsf{toks}_N(1)\end{array}\right\} \texttt{FAA}(\ell, 1) \left\{m.\, (\exists k \geq 0.\, \ell \mapsto k * \mathsf{toks}_N(k)) * 0 \leq m\right\}$$

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

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

To verify the FAA step, we first apply the frame rule WFREE:IRIS-HT-FRAME that strips the irrelevant resources from both the pre- and post-condition. We close the proof by applying IRIS-FAA to verify the remaining triple

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

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

## 5.4 Verifying Wait-Freedom in Lawyer Logic

In this section, we show how an operation is verified against the wait-freedom specification of Definition 11. As an example, we consider `incr` $l$ from Figure 5.1c. Specifically, we verify $\mathsf{WaitFreeSpec}_{\mathit{hasInt}(l)}(\texttt{incr}\, l)$ in the Lawyer logic for any location $l$ (see Section 5.2 for the definition of $\mathit{hasInt}(l)$). Doing so allows us to use Theorem 4 to conclude $\mathit{waitFree}_{\mathit{hasInt}(l)}(\texttt{incr}\, l)$ holds at the meta-level. Most of the verification process amounts to proving the Hoare triples similarly to how it was done in Section 5.3, although the details of program logics are different.

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

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

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

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

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

With the ownership of the entire heap, we will prove the specifications $\mathsf{NoInfExec}(\texttt{incr}\,l)$ and $\mathsf{PresInv}(\texttt{incr}\,l)$. Proving both relies on the fact that $l$ always stores some number. For this, we use a counter module invariant (which is even simpler than one used in Section 5.3):

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

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

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

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

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

The number $\mathbf{F}$ is (an over-approximation of) the upper bound on the execution time of op. The actual specification for op is defined as a *Lawyer Hoare triple*. Lawyer [94] is an instantiation of the Trillium framework, which is in turn an extension of Iris that establishes *refinement* of a program to a transition system. In particular, compared to the triples of Iris introduced in Section 5.3, the Hoare triples in Lawyer additionally enforce that every single step of the program is matched by a transition in the so-called Obligations Model—a transition system that Lawyer uses to represent terminating programs. Proving a Lawyer Hoare triple for a program implies that any execution trace of that program refines some trace of the Obligations Model. As all traces of the Obligation Model are finite, this refinement implies termination of verified programs. Note that the Lawyer triples are additionally annotated with the thread identifier, because the thread identity is important for proving refinement.

The precondition and postcondition of the triple in $\mathsf{NoInfExec}(\mathsf{op})$ consist of the ghost resources of Lawyer logic. The precondition can be read as an assignment of $\mathbf{F}$ barrels of *fuel* to the thread $\tau$, with an indirection via the *phase* $\pi$ that marks the barrels and is assigned to $\tau$. The fuel is represented by the $\mathbf{F} \cdot \mathsf{bar}_\circ \pi$ resource (a separating conjunction of $\mathbf{F}$ many individual barrels $\mathsf{bar}_\circ \pi$). The phase assignment is written in the Lawyer logic as $\mathsf{ph}_\circ \tau\, \pi$. The fuel resource represents the number of remaining steps the program is allowed to take: as we will see later, verifying

[2]The reason why $\mathbf{F}$ cannot be existentially quantified inside the triple is related to the issues of step-indexing which are described in detail by Spies *et al.* [112].

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

every single execution step consumes one barrel of fuel, similar to how every counter increment consumed a token in Section 5.3. We remark that in Lawyer [94], both the fuel and the phase of resources have one extra parameter each. Technically, we assign these parameters the values that are largely irrelevant for this presentation; instead of showing these values, we just mark both resources with a $\circ$ subscript. In the appendix, we argue why this particular specification pattern of $\mathsf{NoInfExec}(\mathtt{op})$, as well as certain restrictions on the operation's invariants, captures the nature of wait-freedom in the Lawyer logic.

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

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

The triple above essentially states that we can spend up to 3 barrels of fuel to complete the execution of $\mathtt{op}\,a$. To prove it, we use the rules[4] of the Lawyer logic. Compared to the proof rules presented in Section 5.3, the main difference is that the rules of Lawyer feature Lawyer-specific resources besides the Iris ones. This is because Lawyer requires matching every step of execution with a transition in the Obligations Model, which in turn requires resources such as fuel. Specifically, the Lawyer rules for beta-reductions and FAA are the following:

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

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

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

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

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

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

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

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

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

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

As we will explain in Section 5.6, such Hoare triples ensure that `op` can be used by an arbitrary client program, and that it would not violate the internal invariants of `op`. Technically, a client program *uses* `op` by providing it a *safe* argument, captured by the predicate $\mathsf{RS}_{\mathbf{V}}(a)$, and expects `op` to return a safe value back. We will explain what it means to be safe and motivate why it is necessary to restrict only to safe values in Section 5.6.

Since $\mathsf{PresInv}(\texttt{incr}\,l)$ does not establish refinement to a model, its proof follows the process described in Section 5.3. We ignore the $\mathsf{RS}_{\mathbf{V}}(a)$ resource, because `incr` ignores the input argument $a$. The rules used to verify beta-reductions and FAA are exactly the same as in Section 5.3 (except for the different type of triple). Similarly to the proof above, we use $\mathsf{cntInv}$ to verify the FAA step. The main difference there is that the return value $v$ is not ignored—it needs to be shown to be safe, *i.e.* $\mathsf{RS}_{\mathbf{V}}(v)$. However, we know that FAA returns an integer, and, as we will see in Section 5.6, all integers are safe, *i.e.*, $\forall n : \mathbb{Z}.\ \vdash \mathsf{RS}_{\mathbf{V}}(n)$.

## 5.5 Case Studies

As we discussed in the Introduction, while the simple definition of wait-freedom that we used in Sections 5.2 and 5.4 works for many simple programs, for more involved programs, this definition needs to be adjusted. Specifically, Definition 10 follows the classical definition of wait-freedom [45]: it requires an operation to progress safely (*i.e.*, without getting stuck) regardless of its input arguments and the number of concurrent operations. However, in practice each of these conditions can affect the safe progress.

In this section, we consider two illustrative examples that motivate two independent extensions to our wait-freedom verification approach. First, we verify a (higher-order) function that gets stuck on a "wrong" argument. Second, we verify a data structure that restricts the number of operations that can run concurrently at a given time. The latter is especially important, as many realistic wait-free data structures [35, 73, 119] rely on this restriction. We note that these changes account for different types of progress properties, not for each individual example; we believe that the case studies exhibiting the same property can be verified with the same specification and the adequacy theorem.

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

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

Consider the higher-order `list_map` function in Figure 5.6a. It applies the given function `f` to all elements of the list $l$. In our language, lists are values satisfying the recursive predicate $\mathsf{isList}$ defined in Figure 5.6b.

Note that if the input function `f` does not run forever for any argument, the partially applied `list_map f` does not run forever for any argument either. However, it might

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

(a) Implementation

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

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

Figure 5.6: Higher-order list mapping function

get stuck even if `f` never does: if its second argument $l$ does not satisfy isList, then `list_map` will get stuck, *e.g.*, on taking projections. Thus, the progress of calls to `list_mapf` is captured by the following variant of Definition 9, with the changes highlighted:

**Definition 12** (Eventual return or stuckness of calls to op)**.** *For an operation* op : *Val, execution trace etr and thread identifier* $\tau$*, we say that all* $\tau$*'s calls to* op *in etr eventually return or get stuck, written* $alwaysReturnsInTrace^{\lightning}(\mathtt{op}, etr, \tau)$*, iff*

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

Similarly, we define possibly-stuck wait-freedom $waitFree^{\lightning}$ as a variant of Definition 10 that uses $alwaysReturnsInTrace^{\lightning}$ instead of $alwaysReturnsInTrace$.

Our approach to proving wait-freedom supports verifying possibly-stuck wait-freedom with minimal changes compared to the ordinary wait-freedom discussed in earlier sections. That is, we prove a variant of Theorem 4 that establishes possibly-stuck wait-freedom of an operation, given that it satisfies the Lawyer specification $\mathsf{WaitFreeSpec}^{\lightning}$, which, similarly to definitions above, is a weakening of WaitFreeSpec that allows the operation to get stuck.

Now, we can apply the approach outlined in Sections 5.2 and 5.4 to prove that `list_mapf` is possibly-stuck wait-free. Crucially, we do so modularly by verifying both `list_map` and `f` independently of one another. Formally, we prove the following higher-order possibly-stuck wait-freedom specification for `list_map`:

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

As in Section 5.4, we prove the two specifications for `list_mapf`. For each of them, we make use of the corresponding specification for `f`. The logical inference rules used in the process are the same used in Section 5.4, with the addition of rules to verify stuck computations, *e.g.*, the following Hoare triple for projecting of the unit value: $\{\mathsf{True}\}^{\lightning}\ \texttt{fst}\,()\ \{P\}^{\tau}$.

Given the (list-map-HO-spec) spec above, together with the fact that ordinary Hoare triples imply possibly-stuck Hoare triples, we can reuse the already proven (possibly-stuck) wait-freedom specifications to derive possibly-stuck wait-freedom of `list_map` partially applied to specific function argument. For example, by weakening the Hoare triple that we proved in Section 5.4, we obtain a proof of $\mathsf{WaitFreeSpec}^{\lightning}_{\mathit{hasInt}(l)}(\texttt{list_map}\,(\texttt{incr}\,l))$ for any location $l$, thus establishing

$$\mathit{waitFree}^{\lightning}_{\mathit{hasInt}(l)}(\texttt{list_map}\,(\texttt{incr}\,l))$$

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

## Restricted Wait-Freedom

Many practical implementations of wait-free algorithms [35, 57, 70, 119] assume an extra condition not accounted for by Definition 10: that the number of concurrent operations has a fixed upper bound. An example of such implementation is shown in Figure 5.7a. This implementation of a concurrent wait-free queue has two uncommon features. First, besides adding and removing elements to the queue, it allows reading the head element without removing it. Second, it optimizes the memory usage by deallocating the head node when it is removed. Supporting both of these features, and simultaneously safety and linearizability, is highly non-trivial. Therefore, the implementation in Figure 5.7a limits the concurrency to just two threads: an enqueuer thread that can either enqueue elements or read the queue head, and a dequeuer thread that can either dequeue elements or read the queue head.

Verifying such algorithms, which we call *restricted wait-free*, requires multiple changes to our approach as we have described so far. First of all, we define two wrapper functions `enqueuer` and `dequeuer` (see Figure 5.7b) which wrap the possible operations of the queue for the enqueuer and dequeuer threads explained above into a single operation respectively—each function checks its argument and dispatches the appropriate queue operation accordingly. Then, for technical reasons that we explain in Section 5.7, we restrict the queue to only store integers. To that end, the `enqueuer` wrapper includes a call to the **`to_int`** primitive (which we have added to $\lambda^{\mathsf{conc}}_{\mathsf{ref}}$ for this example), the semantics of which are as follows:

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

Restricted wait-freedom is stated as a variant of Definition 10, with changes highlighted:[6]

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

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

let read_head_e () =                let deque () =
  let cur_head = !Head in             let cur_head = !Head in
  if (cur_head == !Tail)              if (cur_head == !Tail)
  then None                           then None
  else                                else
    BeingRead := cur_head;              let v = cur_head.val in
    let v =                             OldHeadVal := v;
      if (cur_head == !Head)            Head := cur_head.next;
      then                              let to_free =
        cur_head.val                      if (cur_head == !BeingRead)
      else                                then
        !OldHeadVal                         let old = !FreeLater in
    in Some v                               FreeLater := cur_head;
                                            old
let enqueue v =                           else cur_head
  let dummy =                           in free to_free; Some v
    Node {val: 0, next: null} in
  let new_tail = ref dummy in       let read_head_d () =
  let cur_tail = !Tail in             let cur_head = !Head in
  cur_tail.val := v;                  if (cur_head == !Tail)
  cur_tail.next := new_tail;          then None
  Tail := new_tail                    else Some cur_head.val
```

(a) Original implementation by Jayanti and Petrovic [57] expressed in $\lambda_{\text{ref}}^{\text{conc}}$

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

(b) Wrappers to be verified

Figure 5.7: Single-enqueuer, single-dequeuer queue

**Definition 13** (Restricted wait-freedom)**.** *Given a set of initialized configurations*

*$C \subseteq Config$, we define restricted wait-freedom of list of operations* $\texttt{ops} : List(Val)$ *as*

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

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

For example, restricted wait-freedom of the algorithm in Figure 5.7b is stated using Definition 13 by taking $\texttt{ops} \triangleq [\texttt{enqueuer}, \texttt{dequeuer}]$. A more complicated $n$-enqueuer, single-dequeuer algorithm (also presented in [57]) requires using a list consisting of $n$ enqueuer methods and a single dequeuer method, *i.e.*, $\texttt{ops} \triangleq \texttt{dequeuer} :: repeat(n, \texttt{enqueuer})$.

Verifying restricted wait-freedom requires changes to the specification we had given in Definition 11. We will first give the formal definition and explain the details afterwards.

**Definition 14** (Lawyer specification of restricted wait-freedom)**.** *For a set of initialized configurations $C \subseteq Config$, we define the Lawyer specification of* $\texttt{ops} : List(Val)$ *restricted wait-freedom as follows, for some* $\mathbf{F} : \mathbb{N}$*:*

$$
\begin{array}{l}
\mathsf{WaitFreeRestrSpec}_C(\texttt{ops}) \triangleq \forall c \in C.\ \mathsf{heap_repr}(c) \Rrightarrow \mathsf{mtoks}(\texttt{ops}) * \\
\quad \mathop{\Large *}_{\texttt{op} \in \texttt{ops}} \left( \forall \tau, \pi, a.\ \{\mathbf{F} \cdot \mathsf{bar}_\circ \pi * \mathsf{ph}_\circ \tau\, \pi * \mathsf{mtok}(\texttt{op})\}\ \texttt{op}\ a\ \{\mathsf{ph}_\circ \tau\, \pi * \mathsf{mtok}(\texttt{op})\}^{\tau} \right) * \\
\quad \mathop{\Large *}_{\texttt{op} \in \texttt{ops}} \left( \forall \tau, a.\ \{\mathsf{mtok}(\texttt{op})\}^{\not\downarrow}\ \texttt{op}\ a\ \{v.\, isGroundVal(v) * \mathsf{mtok}(\texttt{op})\}^{\tau} \right) \\
\textit{where}\ \mathsf{mtoks}(l : List(Val)) \triangleq \mathsf{toks}_{\texttt{ops}}(l) \quad \textit{and} \quad \mathsf{mtok}(m : Val) \triangleq \mathsf{mtoks}([m])
\end{array}
$$

In the definition above, $isGroundVal(v : Val)$ states that the value $v$ does not contain any locations or lambda functions. This is related to the restriction where we require the queue to only store integers which we will discuss in Section 5.7. The proposition $\mathsf{mtoks}(l)$ (an abbreviation for $\mathsf{toks}_{\texttt{ops}}(l)$[7]) asserts ownership of the list $l$ of *method tokens*. Similarly to the tokens used in Section 5.3, these are created upon the module initialization and have to be provided upon every call to the corresponding operation; the difference is that a method token is given back after the return to allow subsequent

[7] In fact, the tokens resource $\mathsf{toks}_N(k)$ from Figure 5.4 can be defined for any type of $k$ and $N$ that defines (disjoint) composition and the corresponding inclusion order.

calls. Method tokens satisfy the following laws similar to (tokens-sep) and (tokens-le); here, $\mathsf{count}(m, l)$ counts occurrences of $m$ in $l$:

$$\mathsf{mtoks}(l_1) * \mathsf{mtoks}(l_2) \dashv\vdash \mathsf{mtoks}(l_1 +\!+ l_2)$$

$$\mathsf{mtoks}(l) \vdash \forall m \in l.\, \mathsf{count}(m, l) \leq \mathsf{count}(m, \mathtt{ops})$$

Finally, the adequacy theorem for restricted wait-freedom is similar to Theorem 4:

**Theorem 5** (Adequacy for restricted wait-freedom)**.** *If* $\mathsf{WaitFreeRestrSpec}_C(\mathtt{ops})$ *is provable in Lawyer, then* $\mathit{waitFreeRestr}_C(\mathtt{ops})$ *holds in the meta-logic.*

We use Theorem 5 to establish $\mathit{waitFreeRestr}_{C_Q}([\mathtt{enqueuer}, \mathtt{dequeuer}])$ (where $C_Q$ is some set of allowed initial states for the queue in Figure 5.7a) by verifying the wrappers in Figure 5.7b. We note that despite the fact that termination of queue methods follows almost immediately—they all terminate in constant time—proving their safety is highly non-trivial: reading of the head node might execute concurrently with dequeueing that involves freeing the head. We refer the reader to our Rocq development for the details of the proof.

## 5.6 Proving the Wait-Freedom Adequacy Theorem

As Section 5.4 demonstrates, proving the specification in Definition 11 for a given operation is relatively straightforward. Indeed, a wait-free operation cannot block, or be delayed by other operations running concurrently, thus it must terminate in a finite number of steps. To account for this, the rules of Lawyer logic simply consume one fuel resource upon every step; essentially, the user is mainly responsible for providing enough fuel initially. This applies both to the original specification in Definition 11, and its variants described in Section 5.5. Thus, the non-trivial parts of the proof might only concern safety of the operation.

However, showing that the specification in Definition 11 implies wait-freedom, *i.e.* proving Theorem 4, is quite challenging. Ideally, we would just reuse the existing work on termination verification in Lawyer for proving wait-freedom. That is, we would ideally reuse Lawyer's proof that verifying a program in Lawyer establishes a refinement between the program and an always terminating transition system, which then guarantees that the program terminates. We might hope that this would help us derive termination of all calls to our wait-free operation. However, we face the following challenges:

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

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

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

We address these challenges as follows. First, in Section 5.6 we reduce the problem of proving wait-freedom to showing termination of the closed program. With that, we are able to reuse the Lawyer's adequacy theorem, thus addressing the Challenge 1. Then, in Section 5.6 we employ the well-known method of logical relations to establish "robust safety" of the wait-free operation, *i.e.*, we prove that an arbitrary client does not violate operation's internal invariants. This solves the Challenge 2. Finally, in Section 5.6 we show that the aforementioned reduction in fact allows us to apply the Lawyer logic to generate fuel consumed by an infinitely running program. Together with the robust safety result, it allows us to verify a non-terminating program, which solves Challenge 3.

Finally, we note that Theorem 4 and its variants for possibly-stuck wait-freedom from Section 5.5 are proven in the same way with minimal variations. However, proving Theorem 5 requires non-trivial changes to the logical relation and the way it is used. For space reasons, we refer the reader to technical Rocq development for that. Therefore, the rest of the section only explains how the original Theorem 4 is proven.

### Reducing Wait-Freedom to Termination

The first step in proving the wait-freedom adequacy theorem is to reduce proving wait-freedom to showing termination of the closed program. For that, the proof of Theorem 4 proceeds via a proof by contradiction. Namely, for the operation op and some set of initialized configurations $C \subseteq \mathit{Config}$ we assume there exist an execution trace $\mathit{etr}$, evaluation context $E$, thread $\tau$, and trace index $i$ such that:

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

Note that the only way this is possible is in the case where $\mathit{etr}$ is infinite. Indeed, if the call to op never returns, then by the *schedUntilRet* assumption of *alwaysReturnsInTrace* the calling thread $\tau$ is always eventually scheduled. We will

prove the contradiction by showing that the same set of assumptions simultaneously implies that *etr* must terminate.

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

The key to establishing the refinement is exploiting the assumptions above. Indeed, in Section 5.6 we show how to reuse Lawyer logic and construct a refinement *assuming* the premises above and given the specification for the wait-free operation. In other words, we establish the refinement *only* for execution traces that satisfy the premises above. This is in contrast with the adequacy theorem of Lawyer (and Trillium) which establish the refinement for *all* execution traces. Therefore, we need to generalize the adequacy theorem of Trillium to establish a refinement only when some condition on execution holds. In turn, the fact that this condition holds is propagated to the refinement construction via user-provided "progress resource" described in more detail in Section 5.6. Below, we provide the generalized adequacy theorem of Trillium and highlight the main changes compared to the original theorem.

**Theorem 6** (Conditional adequacy theorem of Trillium)**.** *Consider a transition system $\mathscr{M}$ and a relative image-finite relation $\xi$ on finite traces of $\lambda_{\mathsf{ref}}^{\mathsf{conc}}$ and the model. Then, consider $\mathsf{TI} : FinTrace(\lambda_{\mathsf{ref}}^{\mathsf{conc}}) \to FinTrace(\mathscr{M}) \to iProp$ — a representation of the aforementioned finite traces in the Iris logic. Then, consider a predicate $F$ on finite execution traces. Finally, consider a "progress resource" $\mathsf{PR}_F : FinTrace(\lambda_{\mathsf{ref}}^{\mathsf{conc}}) \to iProp$ that satisfies a number of laws described in appendix[91].*

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

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

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

We refer the reader to Trillium [118] for definitions of AlwaysHolds and $\hat{\xi}$; the latter is essentially the refinement we are looking for. Intuitively, the three resources in the Iris premise of Theorem 6 correspondingly describe *what* is the current state of the execution, *why* this state representation implies refinement at the meta-level, and *how* this representation is preserved on every step of the execution.

We prove the finiteness of our *etr* by applying Theorem 6 and establishing the refinement to the Obligations Model. For that, we choose the trace interpretation, the starting model state $\mathsf{m}$, and the refinement relation that the Lawyer's adequacy theorem does when it uses Trillium's adequacy theorem (the latter being slightly

adjusted — see our Rocq implementation for details). We defer the choice of the predicate $F$ and the progress resource to Section 5.6; we only note that the chosen $F$ indeed holds for all prefixes of *etr*. Thus, it remains to prove the Iris premise of Theorem 6. There, $\mathsf{TI}(etr[0], \mathsf{m})$ and $\mathsf{AlwaysHolds}(\xi, etr[0], \mathsf{m})$ are proved similarly to how it is done in Lawyer. We will explain how to establish the progress resource in Section 5.6. Thus, using the Trillium's adequacy theorem we prove that *etr* is finite, and thus derive a contradiction.

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

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

However, it turns out that verifying the operation's client can be split into two steps. The first step is proving the *robust safety* of the operation, *i.e.* the fact that any operation's client preserves operation's internal invariants. The next step, described further in Section 5.6, is showing that any execution of a client refines the Obligations Model of Lawyer.

To prove robust safety, we employ the well-known idea of logical relations [34, 116]. This approach is based on proving, for a given expression $e$, an Iris proposition $\mathsf{RS}_{\mathbf{E}}(e)$; the predicate $\mathsf{RS}_{\mathbf{E}}(\cdot)$ is called the *expression relation*. The expression relation intuitively captures that any execution of $e$ preserves all invariants in the system (including those of the wait-free operation), but it may get stuck. Crucially, the expression relation is proven to be closed under program compositions, *e.g.*, if both $e_1$ and $e_2$ are in the expression relation, so is the expression $e_1\ e_2$, *i.e.*, calling the function $e_1$ with argument $e_2$—intuitively, the worst that can happen here is that $e_1$ is not a function, in which case $e_1\ e_2$ gets stuck, which is allowed. This compositionality of the expression relation is the reason why, to establish robust safety of the operation, it is sufficient to prove $\mathsf{RS}_{\mathbf{E}}(e)$ for all clients $e$ of the operation. In Section 5.4 we described a PresInv Hoare triple that states a property similar to $\mathsf{RS}_{\mathbf{E}}(\cdot)$. Indeed, expression relation is defined similarly to PresInv (we explain $\mathsf{RS}_{\mathbf{V}}(v)$ below):

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

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

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

**Definition 15** (Value relation)**.**

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

Note that the last two cases of this definition reference $\mathsf{RS}_{\mathbf{V}}(v')$ for some value $v'$ which might be structurally larger than the one being considered. The soundness of such definitions is guaranteed by the guarded recursion of Iris; we refer the reader to [63] for more details.

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

However, an $e$ that actually calls a wait-free operation `op` requires some extra work. Indeed, values produced during `op`'s execution might not satisfy the value relation (*e.g.* locations used by `op` internally are covered by `op`'s own invariants), and also `op` might use pointer arithmetic (*e.g.* the queue in Figure 5.7a uses it for accessing `Node` structure fields). Thus, we cannot automatically derive expression relation for calls to `op`—instead, the user must show it. And indeed, the $\mathsf{PresInv}(\texttt{op})$ part of `op`'s wait-free specification is exactly the lambda-expression case of Definition 15, stating that it is always safe to call `op` with any argument. Another way to think about this is that showing $\mathsf{PresInv}(\texttt{op})$ is also required because it ensures that the operation does not expose its own internal state, which would of course not be safe.

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

**Theorem 7** (Fundamental theorem)**.** *Let $e$ be an expression that contains neither any Loc literals nor any pointer arithmetic, and whose free variables are $\vec{xs}$. Given any list of values $\vec{vs}$ such that $|\vec{vs}| = |\vec{xs}|$. Then the following holds:*

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

An `op`'s client can be represented as a thread pool made of expressions into which `op` is substituted (see the first premise of Definition 10). To prove robust safety of

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

op, we apply Theorem 7 and provide $\mathsf{PresInv}(\mathtt{op})$ to satisfy its premise. Thus, we establish the following:

**Theorem 8** (Robust safety of the wait-free operation)**.** *Let $\mathscr{E}$ be a thread pool such that its every element e has a single free variable x, and contains neither Loc literals, nor any pointer arithmetic. Assume that* $\mathsf{WaitFreeSpec}_C(\mathtt{op})$ *holds for some C. Then the following holds:*

$$\vdash \mathop{\Large\ast}_{e\in\mathscr{E}} \mathsf{RS}_{\mathbf{E}}(e[\mathtt{op}/x])$$

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

In Section 5.6, we have reduced proving wait-freedom to that of termination, which we establish by applying Theorem 6 and refining a given execution to a trace of the Obligations Model of Lawyer. However, that reduction, by contradiction, considers an *infinite* execution, whereas the main property of the Obligations Model is the finiteness of its traces. As the Challenge 3 states, the Lawyer logic that establishes refinement to the Obligations Model was designed to only verify terminating programs. Here, we briefly describe the solution that allows us to repurpose the Lawyer logic for verifying non-terminating programs. Due to lack of space we refer the reader to our Rocq formalization [93] for the details.

The central part in establishing refinement with Theorem 6 is the progress resource. Intuitively, the resource $\mathsf{PR}_F(etr)$ means that if the finite execution trace *etr* is currently refined by some trace of the Obligations Model, and if it takes a step into a new trace $etr'$ such that $F(etr')$ holds, then $etr'$ will also refine some Obligations Model trace, all the invariants are preserved, and $\mathsf{PR}_F(etr')$ is reestablished. Thus, the progress resource for a singleton trace $\mathsf{PR}_F(c)$ essentially means that any execution starting from the configuration $c$ is refined by a trace of the Obligations Model, as long as it does not violate the predicate $F$.

Crucially, the user is free to choose an arbitrary progress resource, as long as it satisfies a number of laws listed in the appendix [91]. The original adequacy theorem of Lawyer can be stated as a corollary of Theorem 6 where the progress resource ignores the predicate $F$, and for the starting configuration it consists of Hoare triples for every thread present in that configuration. In our case, we cannot prove such triples, as we consider non-terminating client programs, and the Lawyer logic is designed to establish termination.

Thus, the progress resource for wait-freedom consists of two parts: one that guarantees preservation of invariants and another that ensures the refinement of the Obligations Model. The former is given by the robust safety result of Theorem 8; the latter is more elaborate.

The main part for establishing a refinement of the Obligations Model is providing a barrel of fuel on every execution step. For that, we exploit the assumptions on the execution that were obtained during the reduction in Section 5.6. Namely, that our progress resource is defined as $F(etr) \triangleq \mathit{infCallPrefix}(E,\tau,i,etr)$, where $E,\tau$ and $i$ are fixed during the reduction. We refer the reader to Rocq development [93] for its

formal definition. Intuitively, this predicate means that *etr* is a possible finite prefix of some infinite trace in which thread $\tau$ has an infinite call to `op` under evaluation context $E$ at index $i$. Now, knowing $i$, we can pre-allocate $i$ barrels of fuel in the initial progress resource, so that the refinement can be preserved for the first $i$ steps before the infinite call.

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

To provide fuel to all other threads, we employ a trick that allows us to *generate fuel indefinitely*. That is, we use the Lawyer logic's obligations mechanism originally designed for verifying blocking concurrency. We refer the reader to Lawyer [94] for more details; in short, threads can wait for other threads' actions by generating fuel, as long as this waiting is not circular. In our case, we can assign the calling thread $\tau$ an obligation that is never fulfilled which does not correspond to any action, but merely enables the fuel generation mechanism. This lets all other threads to wait for this obligation to be fulfilled. By assumption, $\tau$ never terminates, and thus the other threads can always receive fuel. Of course, this trick is only possible because we started the proof in Section 5.6 by contradiction—under normal circumstances, a thread with an obligation eventually fulfills it, which disables fuel generation.

## 5.7 Discussion and Limitations

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

When verifying the queue implementation in Section 5.5, we introduced a seemingly unnecessary constraint. Namely, the wrapper `enqueuer` in Figure 5.7b only allows the integer values to be enqueued. Note that the original implementation in Figure 5.7a never inspects the enqueued values and thus does not rely on this constraint. Yet, in short, lifting this constraint would imply adding an extra premise to the precondition of NoInfExec which must hold for the argument — but since we show termination of calls to an operation with *any* argument, we would not be able to prove it when using NoInfExec in the adequacy proof.

Before proceeding to details of this constraint, we explain its scope. Despite arising when verifying restricted wait-freedom in Section 5.5, the issue is not related to tokens, or the restricted wait-freedom in general. However, it arises with all data structures that contain methods that return values previously stored by other methods (queues, stacks, linked lists *etc.*). In our context, such structures happen to be restricted wait-free.

Recall that establishing robust safety of `dequeuer` in Figure 5.7a using Theorem 8 requires proving the PresInv(`dequeuer`) specification, which is the lambda-expression case of Definition 15. (For technical reasons, Definition 14 requires a stronger specification from which PresInv(`dequeuer`) is derivable, but the issue

would still hold without this stronger specification.) Note that when `dequeuer` returns a dequeued value, it cannot, by itself, prove that it is safe-to-use. For this to hold, the queue invariant must enforce that all values stored are safe-to-use by storing the $\mathsf{RS}_{\mathbf{V}}(v)$ resources for every stored value $v$. To maintain this invariant, `enqueuer` must only call `enqueue` with safe-to-use values — and it indeed does, as only integers are being enqueued.

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

To lift this limitation, one has to exploit the fact that a client program satisfying the conditions of Theorem 8 can only produce safe-to-use values as it executes (except when the wait-free operation is in progress). We leave establishing this formally to future work.

### Proving linearizability of wait-free data structures

An important criteria for the correctness of a concurrent data structure is *linearizability* [46]. A data structure is linearizable if whenever operations on that structure are run concurrently, each of them appears to take effect atomically, even if they actually take multiple steps to finish. Linearizability makes it easier to reason about the data structure as one does not have to consider all possible interleavings of the operations—one can simply reason as though these operations happen one after another. It is thus not surprising that many wait-free data structures are designed to be linearizable [35, 57, 70, 119]. Therefore, it is desirable to verify both wait-freedom and linearizability for wait-free data structure.

In fact, proving linearizability with program logics is well-studied [13, 53, 62, 100]. Birkedal *et al.* [13] have formally shown that linearizability is internalized by specifications given in terms of the so-called *logically atomic Hoare triples* [62]. Moreover, the Lawyer approach [94] that we build upon provides a liveness-aware version of the logically atomic specification. However, even simple wait-free implementations contain what is known as *future-dependent linearization points* [122] where the point at which linearization takes place within an operation can depend on how the other operations that started afterwards execute. Moreover, the linearization point of an operation might happen inside another operation that runs concurrently with it in a different thread. It is the case for the queue in Figure 5.7a. Consider the scenario where the dequeuer thread runs a number of `dequeue` operations one by one, while the enqueuer thread is running the single `read_head_e` operation. In that case, the linearization point of `read_head_e`, if the comparison after the second read of `Head` fails, will occur *when the last* `dequeue` *operation writes to* `OldHeadVal`.

Iris supports verifying future-dependent linearizability using so-called prophecy variables [64]. However, as is, Lawyer does not support prophecy variables because Trillium [118] does not. Thus, verifying both wait-freedom and linearizability in our framework would require extending the Lawyer logic and the Trillium logic first.

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

Some data structures provide both lock-free and wait-free methods. For example, an implementation of a counter can include a lock-free increment from Figure 5.1b and a wait-free `read` method that simply reads the current counter value in a single step. Thus, in general, wait-freedom is a property of a single method, not an entire data structure. Our approach could also be used for verifying wait-freedom in such a setting as well. First, the meta-level definition of wait-freedom should mention both a wait-free operation, and a number of lock-free operations that may run concurrently with it (similarly to how Definition 13 applies to the list of wait-free operations). As for the specifications, the approach suggested by Total TaDA [101] applies: the operations running concurrently with the wait-free one only need to preserve invariants. With that, we can prove a generalization of Theorem 4 that establishes wait-freedom of an operation $\mathtt{op}$ given $\mathsf{WaitFreeSpec}(\mathtt{op})$ and proofs of $\mathsf{PresInv}(\mathtt{op}')$ for every lock-free operation $\mathtt{op}'$ running concurrently with $\mathtt{op}$.

## 5.8 Related Work

All prior solutions verify wait-freedom for first-order languages. These solutions assume, in one way or another, that the client of a wait-free operation is safe, *i.e.* that it does not violate the data structure's internal invariants. Instead, we prove once and for all that all possible clients in our higher-order programming language satisfy this property via a logical relations argument as explained in Section 5.6. Moreover, most of the prior solutions only support unrestricted wait-freedom, lack case studies, and are not mechanized.

### Program Logics for Wait-Freedom

**Total TaDA** The program logic of [101] supports verification of non-blocking program properties, including unrestricted wait-freedom. Proving a program specification in that logic establishes a meta-level judgement about program execution. This judgement is defined as a least fixed point and thus guarantees the program's termination. The non-trivial (from liveness perspective) rules of the Total TaDA logic allow verification of while loops and recursive functions by choosing a decreasing measure of termination. Moreover, their specifications are capable of expressing linearizability, which we do not consider. This approach is demonstrated by verifying wait-freedom of `read` operation of a counter.

Total TaDA assumes that the operation's client is safe by incorporating a rely relation [61] into the aforementioned meta-level judgement established for the operation. The only way to ensure that the client respects the rely relation is to verify it; thus their definition of wait-freedom does not quantify over arbitrary clients as ours does. However, they support clients verified with the non-total TaDA [100] logic. Contrary to that, our approach supports arbitrary client programs (up to assumptions of Theorem 7). Total TaDA is not mechanized.

**Lili** The approach of [80] verifies liveness of both blocking and non-blocking operations of concurrent objects, as well as their linearizability. In that work, (unrestricted) wait-freedom is treated as starvation-freedom (a property usually applied to blocking algorithms) without the fair scheduling assumption. The Lili logic is parametric, and by instantiating it in a specific way one can exclude rules that justify threads being delayed and/or blocked by each other. This is similar to how Lawyer proves *e.g.* termination under an unfair scheduler with a more restrictive instantiation of the Obligations Model.

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

The Lili logic is neither higher-order nor mechanized. Moreover, neither Lili, nor the lock-free fragment of its predecessor [81] has any case studies on wait-freedom verification.

**Automatic tool for verifying non-blocking algorithms** The work of [36] verifies progress properties of non-blocking algorithms. They notice that progress of these algorithms can be established by identifying a set of ordered abstract actions. An action at a higher level cannot be executed infinitely often unless there is an action of a lower level which is also executed infinitely often. In this setting, (unrestricted) wait-freedom corresponds to discrete orders, *i.e.* no action can ever be executed infinitely often. In the absence of non-trivial orders between actions and unbounded `while` loops, Gotsman *et al.* [36] reduce verifying wait-freedom to verifying safety.

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

Gotsman *et al.* [36] present no wait-freedom case studies. They explain that their approach might not apply to non-trivial wait-free operations due to the limitations of their tool. As an example, they mention the wait-free `contains` operation of the

concurrent set algorithm by [124]. The `while` loop in that operation traverses a sorted list of integers up to a given number. As the list can be concurrently modified during the traversal, termination of such a loop relies on the list being sorted. The tool by [36] cannot exploit this fact, whereas in our approach the termination of such loops can be shown by providing the amount of fuel proportional to the given number.

Moreover, their tool has limited safety reasoning capabilities and thus cannot verify examples like Figure 5.7a where the safety argument is highly non-trivial. The tool is automated, but the presented theory is not mechanized in a proof assistant.

**Wait-freedom of "fast-path-slow-path" in VST** The work of [99] considers non-blocking algorithms that are instances of a specific variation of the "fast-path-slow-path" approach [70]. This approach aims to design efficient wait-free data structures by combining a lock-free implementation (which is often faster, but does not have a wait-free progress guarantee) and a wait-free implementation (slower, but has guaranteed progress).

The authors of [99] establish a pen-and-paper proof that such algorithms, if safe, are indeed wait-free, and provide an extension of the VST logic [11] to verify their safety, and thus also wait-freedom. This approach limits the number of concurrent threads, and therefore it can only be used to establish restricted wait-freedom. Similar to Total TaDA and Gotsman *et al.* [36], such approach requires the user to verify the entire program.

As a case study, Peterson *et al.* [99] verify wait-freedom of the queue by Kogan and Petrank [70], which is in turn an optimization of the queue by Kogan and Petrank [69]. In the implementation by [70], a thread helps up to $N$ other concurrent operations before performing its own operation (where $N$ is the maximal number of threads). We believe that our approach can also support verifying this case study by requiring an amount of fuel proportional to $N$.

## Universal Constructions

Designing a wait-free implementation for every possible data structure is tedious, and therefore a question arises as to whether it is possible to design a universal wait-free construction. And indeed, that is possible. These are algorithms that turn any *sequential* data structure implementation into a (restricted) wait-free one. The early universal constructions such as [45] are considered impractical due to their high time and space complexity of the resulting wait-free implementations. Luckily, this is less of an issue for more recent constructions mentioned below. However, none of these constructions has a mechanized proof of wait-freedom.

Therefore, it is interesting whether our solution can be used to verify such constructions. To do so, we would need to prove a specification for the sequential implementation, and use it together with a general proof of the universal construction in question — a process similar to how we verified possibly-stuck wait-freedom of `list_map`$(\texttt{incr}\, l)$ in Section 5.5. Therefore, we believe that our approach can be applied to universal constructions as well.

Examples of modern wait-free universal constructions include P-Sim [29] (along with its extensions such as [30]) and CX [19]. The time complexity of these approaches is proportional to the number of concurrent operations, and the execution time of the provided sequential implementation. Given that these parameters are known, we believe that our approach can verify restricted wait-freedom of the resulting implementations, as the required amount of fuel is known in advance.

## 5.9 Conclusion and Future Work

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

There are multiple notable directions of future work. First, we aim to extend our approach to verify related properties, such as linearizability and bounded wait-freedom [17]. It would also be interesting to verify one of the state-of-the-art universal constructions for wait-freedom. Finally, we hope to be able to improve the usability of our approach by deriving the two specifications needed by Definition 11 from a single, stronger specification that implies both.

## 5.10 Appendix

### Lawyer Specifications of Non-Blocking Properties

In Section 5.4, we claimed that the specification $\mathsf{NoInfExec}(\mathtt{op})$ captures that all calls to op must terminate. We furthermore claimed that $\mathsf{PresInv}(\mathtt{op})$ implies that op does not rely on any invariants that involve Lawyer resources. Thus, according to Liang and Feng [80], these conditions together imply that op is wait-free as it must never block itself, nor can it delay other threads. Below we will explain how these specifications prohibit blocking and delaying, and discuss the changes needed in our approach in order to support other non-blocking properties.

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

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

Neither its precondition nor its postcondition mentions any obligation resource $\mathsf{obls}\,\tau\,\mathcal{O}$ of Lawyer. This resource indicates that the thread $\tau$ holds a set $\mathcal{O}$ of *obligations* that it must fulfill, *e.g.*, setting a shared flag that other threads can wait for. In Lawyer, a thread having an obligation allows the other threads to busy-wait for this obligation to be fulfilled. In practice, waiting on another thread's obligation means that the waiting thread can *generate* fuel to be spent during its busy-waiting. To prevent circular waiting, a thread can only generate fuel by presenting its own set of obligations, and crucially proving that all its obligations are "ordered after" the obligation (of the other thread) that it waits for.[10] Thus, $\mathsf{NoInfExec}(\mathtt{op})$ not including obligations resources means that it is non-blocking, because it cannot generate fuel by waiting for other threads' obligations, and thus it can only spend the fuel that is explicitly provided to it in $\mathsf{NoInfExec}(\mathtt{op})$.

A delay in a lock-free algorithm is a situation when the successful completion of one operation makes another operation running concurrently with it fail and have to retry. An example of it can be seen in Figure 5.1b where a successful **`CAS`** in one thread can cause another thread to fail its own **`CAS`** which forces it to attempt to increment again by making a recursive call. To prohibit delays, our approach disables the fuel sharing mechanism of Lawyer. This mechanism, introduced by [52], requires a thread that delays others to provide them with extra fuel: since the delayed thread will retry its operation, it will need more fuel. In Lawyer (see [94, Delaying example]), the fuel is shared via an invariant that is accessible by both the delaying thread, and the delayed thread. To disable fuel sharing, we simply require that invariants of the wait-free operation do not contain any Lawyer resources.[11]

Besides wait-freedom, there are two other non-blocking properties: obstruction-freedom [44] and lock-freedom. The former only requires an operation to progress if it is delayed by other operations a finite number of times. When this condition is violated, an obstruction-free operation might block. Thus, we believe that adapting our approach to verifying obstruction-freedom could be achieved by adding obligation resources to the specification. For lock-freedom, one could follow the approach of Lawyer and enable fuel sharing, *i.e.* by lifting the aforementioned restriction on invariants. However, the soundness of fuel sharing in Lawyer relies on the "phases" mechanism [52], which only allows sharing fuel among the threads that have the same ancestor in the fork tree. The definition of lock-freedom does not place any restrictions on threads' ancestry, and therefore this restriction too must be lifted from the fuel sharing mechanism.

## Definitions Used in the Conditional Adequacy Theorem

In Section 5.6, we used the notion of a progress resource. Below, we define it formally. In the Rocq development, we provide a slightly more general version which accounts for threads getting stuck and/or forking. Also, note that in Section 5.6 we treated

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

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

progress resource as a predicate with one argument, as its second argument was not relevant for the discussion.

**Definition 16** (Progress resource)**.** *Consider a labeled transition system $\mathscr{M}$. For it, consider a trace interpretation predicate and a trace invariant predicate*

$$\mathsf{TI}, \mathsf{TrInv} : \mathit{FinTrace}(\lambda_{\mathsf{ref}}^{\mathsf{conc}}) \to \mathit{FinTrace}(\mathscr{M}) \to \mathit{iProp}$$

*where the latter is persistent. Then, consider a postcondition predicate*

$$\mathsf{Post} : \mathsf{Thread} \to \mathit{Val} \to \mathit{iProp}$$

*Finally, consider a predicate on finite execution traces*

$$F : \mathit{FinTrace}(\lambda_{\mathsf{ref}}^{\mathsf{conc}}) \to \mathit{Prop}$$

*With that, we call any*

$$\mathsf{PR}_F : \mathit{FinTrace}(\lambda_{\mathsf{ref}}^{\mathsf{conc}}) \to \mathit{List}(\mathit{Val} \to \mathit{iProp}) \to \mathit{iProp}$$

*a* progress resource *if it satisfies the two following laws:*

$$\begin{array}{l}\forall \mathit{etr} : \mathit{FinTrace}(\lambda_{\mathsf{ref}}^{\mathsf{conc}}), \mathit{mtr} : \mathit{FinTrace}(\mathscr{M}), \Phi s : \mathit{List}(\mathit{Val} \to \mathit{iProp}).\\ \quad F(\mathit{etr}) \mathrel{-\!\!*} \mathsf{PR}_F(\mathit{etr}, \Phi s) \mathrel{-\!\!*} \mathsf{TI}(\mathit{etr}, \mathit{mtr}) \Rrightarrow \\ \quad \mathsf{postsOf}(\mathit{etr}.\mathsf{last}.1) * \mathsf{TI}(\mathit{etr}, \mathit{mtr}) * (\mathsf{postsOf}(\mathit{etr}.\mathsf{last}.1) \mathrel{-\!\!*} \mathsf{PR}_F(\mathit{etr}, \Phi s))\end{array} \tag{prPosts}$$

$$\begin{array}{l}\forall \mathit{etr} : \mathit{FinTrace}(\lambda_{\mathsf{ref}}^{\mathsf{conc}}), \mathit{mtr} : \mathit{FinTrace}(\mathscr{M}), \Phi s : \mathit{List}(\mathit{Val} \to \mathit{iProp}),\\ \quad c' : \mathit{Config}, \tau : \mathsf{Thread}.\\ \quad \mathit{etr}.\mathsf{last} \xrightarrow{\tau} c' \mathrel{-\!\!*} \mathsf{TI}(\mathit{etr}, \mathit{mtr}) \mathrel{-\!\!*} \mathsf{TrInv}(\mathit{etr}) \mathrel{-\!\!*} \mathsf{PR}_F(\mathit{etr}, \Phi s) \mathrel{-\!\!*} F(\mathit{etr} \overset{\tau}{::} c')\\ \quad {}_{\top}\Rrightarrow_{\emptyset}\ (\Rrightarrow_{\emptyset} \triangleright)^{|\mathit{etr}|+1}\ {}_{\emptyset}\Rrightarrow_{\top}\\ \exists \delta : \mathtt{State}(\mathscr{M}), \ell : \mathtt{Label}(\mathscr{M}).\ \mathsf{TI}(\mathit{etr} \overset{\tau}{::} c', \mathit{mtr} \overset{\ell}{::} \delta) * \mathsf{TrInv}(\mathit{etr} \overset{\tau}{::} c', \mathit{mtr} \overset{\ell}{::} \delta) *\\ \qquad\qquad \mathsf{PR}_F(\mathit{etr} \overset{\tau}{::} c', \Phi s +\!\!+ \mathsf{newPosts}(\mathit{etr}.\mathsf{last}.1, c'.1))\end{array} \tag{prTakeStep}$$

Above, $\mathit{tr} \overset{y}{::} x$ denotes extending a finite trace $\mathit{tr}$ with a label $y$ and state $x$.

The two definitions related to postconditions are formally defined in the Rocq development:

- $\mathsf{postsOf}(\mathscr{E} : \mathit{ThreadPool})$ gives postconditions of threads of $\mathscr{E}$ that have terminated,
- $\mathsf{newPosts}(\mathscr{E}_1, \mathscr{E}_2)$ adds postcondition predicates to threads forked when thread pool $\mathscr{E}_1$ has stepped into $\mathscr{E}_2$

# Bibliography


[1] Chapter 17. Threads and Locks. https://docs.oracle.com/javase/specs/jls/se10/html/jls-17.html#jls-17.4.5. 4

[2] Core language — The Rocq Prover 9.1.0 documentation. https://rocq-prover.org/doc/V9.1.0/refman/language/core/index.html. 6

[3] [JDK-8256833] [concurrency-interest] ConcurrentLinkedDeque is non-linearizable - Java Bug System. https://bugs.openjdk.org/browse/JDK-8256833. 3

[4] The Lean Language Reference. https://lean-lang.org/doc/reference/latest/. 6

[5] Logsem/lawyer: Rocq mechanization of Lawyer liveness logic and its extension for wait-freedom. https://github.com/logsem/lawyer, . 29

[6] Logsem/trillium at trillium_wfree. https://github.com/logsem/trillium/tree/trillium_wfree, . 29

[7] Multi-threaded executions and data races (since C++11) - cppreference.com. https://en.cppreference.com/w/cpp/language/multithread.html. 4

[8] Weakest preconditions for total program correctness (!65) · Merge requests · Iris / Iris · GitLab. https://gitlab.mpi-sws.org/iris/iris/-/merge_requests/65, September 2017. 8

[9] Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Trans. Program. Lang. Syst.*, 17(3):507–535, May 1995. ISSN 0164-0925. doi: 10.1145/203095.201069. 36

[10] A. Appel and David A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *TOPL*, September 2001. doi: 10.1145/504709.504712. 19, 36

[11] Andrew W. Appel. Verified software toolchain. In Gilles Barthe, editor, *Programming Languages and Systems*, pages 1–17, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-19718-5. 36, 37, 107

[12] Lars Birkedal and Aleš Bizjak. Lecture notes on iris: Higher-order concurrent separation logic. *Lectures notes, September*, 2020. 9, 11, 13, 57, 60, 62, 82, 86, 87, 88

[13] Lars Birkedal, Thomas Dinsdale-Young, Armaël Guéneau, Guilhem Jaber, Kasper Svendsen, and Nikos Tzevelekos. Theorems for free from separation logic specifications. *Proc. ACM Program. Lang.*, 5(ICFP), August 2021. doi: 10.1145/3473586. 30, 104

[14] Fabian Bläse. Non-blocking synchronization in operating systems, 2021. 78

[15] Pontus Boström and Peter Müller. Modular Verification of Finite Blocking in Non-terminating Programs. *ECOOP*, January 2015. doi: 10.4230/LIPICS.ECOOP.2015.639. 8, 38

[16] Marius Bozga, Radu Iosif, and Filip Konecny. Deciding Conditional Termination, August 2014. 7

[17] Hagit Brit and Shlomo Moran. Wait-freedom vs. bounded wait-freedom in public data structures (extended abstract). In James H. Anderson, David Peleg, and Elizabeth Borowsky, editors, *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing, Los Angeles, California, USA, August 14-17, 1994*, Podc '94, pages 52–60, New York, NY, USA, 1994. ACM. ISBN 0-89791-654-9. doi: 10.1145/197917.197950. 108

[18] Hong-Yi Chen, C. David, D. Kroening, P. Schrammel, and Björn Wachter. Bit-Precise Procedure-Modular Termination Analysis. *ACM Transactions on Programming Languages and Systems*, December 2017. doi: 10.1145/3121136. 7

[19] Andreia Correia, Pedro Ramalhete, and Pascal Felber. A wait-free universal construction for large objects, February 2020. 10, 108

[20] Patrick Cousot and Radhia Cousot. An abstract interpretation framework for termination. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 245–258, Philadelphia PA USA, January 2012. ACM. ISBN 978-1-4503-1083-3. doi: 10.1145/2103656.2103687. 9

[21] Yuxin Deng and Davide Sangiorgi. Ensuring termination by typability. *Information and Computation*, July 2006. doi: 10.1016/J.IC.2006.03.002. 7

[22] Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Communications of The Acm*, 22(8):465–476, August 1979. ISSN 0001-0782. doi: 10.1145/359138.359142. 42

[23] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. *European Conference on Object-Oriented Programming*, June 2010. doi: 10.5555/1883978.1884012. 17

[24] Thomas Dinsdale-Young, Philippa Gardner, and Mark Wheelhouse. Abstraction and refinement for local reasoning. *Lecture Notes in Computer Science*, August 2010. doi: 10.1007/978-3-642-15057-9_14. 16, 17

[25] Thomas Dinsdale-Young, L. Birkedal, P. Gardner, Matthew J. Parkinson, and Hongseok Yang. Views: Compositional reasoning for concurrent programs. *POPL*, January 2013. doi: 10.1145/2429069.2429104. 17

[26] Emanuele D'Osualdo, Julian Sutherland, Azadeh Farzan, and Philippa Gardner. TaDA Live: Compositional Reasoning for Termination of Fine-grained Concurrent Programs, December 2021. 9, 23, 24, 25, 26, 38, 64

[27] Derek Dreyer, Georg Neis, and Lars Birkedal. The impact of higher-order state and control effects on local relational reasoning. In Paul Hudak and Stephanie Weirich, editors, *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 143–156. ACM, 2010. doi: 10.1145/1863543.1863566. 80

[28] Justus Fasse and Bart Jacobs. A flexible specification approach for verifying total correctness of fine-grained concurrent modules, December 2023. 9, 10, 23, 25, 27, 36, 37, 38, 62, 63, 68, 71

[29] P. Fatourou and Nikolaos D. Kallimanis. A highly-efficient wait-free universal construction. *SPAA '11*, June 2011. doi: 10.1145/1989493.1989549. 10, 108

[30] Panagiota Fatourou, Nikolaos D. Kallimanis, and Eleni Kanellou. An efficient universal construction for large objects. In Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller, editors, *23rd International Conference on Principles of Distributed Systems (OPODIS 2019)*, volume 153 of *Leibniz International Proceedings in Informatics (Lipics)*, pages 18:1–18:15, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-133-7. doi: 10.4230/LIPIcs.OPODIS.2019.18. 108

[31] R. W. Floyd. Assigning meaning to programs. January 1967. 7

[32] D. Frumin, R. Krebbers, and L. Birkedal. ReLoC. *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, July 2018. doi: 10.1145/3209108.3209174. 19

[33] Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. Simuliris: A separation logic framework for verifying concurrent program optimizations. *Proc. ACM Program. Lang.*, January 2022. doi: 10.1145/3498689. 8, 64

[34] {Aïna Linn} Georges. Designing and proving robust safety of efficient capability machine programs, July 2023. 81, 100

[35] Seep Goel, Pooja Aggarwal, and Smruti R. Sarangi. A wait-free stack. In Nikolaj S. Bjørner, Sanjiva Prasad, and Laxmi Parida, editors, *Distributed Computing and Internet Technology - 12th International Conference, ICDCIT 2016, Bhubaneswar, India, January 15-18, 2016, Proceedings*, Lecture Notes in Computer Science, pages 43–55. Springer, 2016. doi: 10.1007/978-3-319-28034-9_6. 92, 94, 104

[36] Alexey Gotsman, Byron Cook, Matthew J. Parkinson, and Viktor Vafeiadis. Proving that non-blocking algorithms don't block. *ACM-SIGACT Symposium on Principles of Programming Languages*, January 2009. doi: 10.1145/1480881.1480886. 10, 106, 107

[37] Simon Oddershede Gregersen, Alejandro Aguirre, Philipp G. Haselwarter, Joseph Tassarotti, and Lars Birkedal. Almost-Sure Termination by Guarded Refinement. *Proceedings of the ACM on Programming Languages*, 8(ICFP): 203–233, August 2024. ISSN 2475-1421. doi: 10.1145/3674632. 9

[38] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. *SIGPLAN Not.*, 50(1):595–608, January 2015. ISSN 0362-1340. doi: 10.1145/2775051.2676975. 64

[39] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. *USENIX Symposium on Operating Systems Design and Implementation*, November 2016. 8

[40] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. Certified concurrent abstraction layers. *Sigplan Notices*, December 2018. doi: 10.1145/3296979.3192381. 8, 36, 64

[41] J. Hamin and B. Jacobs. Transferring Obligations Through Synchronizations. *European Conference on Object-Oriented Programming*, January 2019. doi: 10.4230/LIPICS.ECOOP.2019.19. 25

[42] Jafar Hamin and Bart Jacobs. Deadlock-Free Monitors. *Lecture Notes in Computer Science*, April 2018. doi: 10.1007/978-3-319-89884-1_15. 10

[43] Hugo Herbelin, Pierre-Marie Pédrot, coqbot, Gaëtan Gilbert, Maxime Dénès, letouzey, Emilio Jesús Gallego Arias, Matthieu Sozeau, Théo Zimmermann, Enrico Tassi, Jean-Christophe Filliatre, Pierre Roux, Guillaume Melquiond, Jason Gross, Arnaud Spiwack, barras, Pierre Boutillier, Vincent Laporte, Jim Fehrle, Stéphane Glondu, Yves Bertot, Jasper Hugunin, Clément Pit-Claudel,

Ali Caglayan, forestjulien, Pierre Courtieu, Yann Leray, Frédéric Besson, Pierre Rousselin, and MSoegtropIMC. Rocq-prover/rocq: Rocq 9.2.0. doi: 10.5281/zenodo.19256047. 11, 18

[44] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *23rd International Conference on Distributed Computing Systems, 2003. Proceedings.*, pages 522–529, Providence, Rhode Island, USA, 2003. IEEE. ISBN 978-0-7695-1920-3. doi: 10.1109/ICDCS.2003.1203503. 10, 109

[45] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991. ISSN 0164-0925. doi: 10.1145/114005.102808. 78, 85, 92, 107

[46] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990. ISSN 0164-0925. doi: 10.1145/78969.78972. 62, 104

[47] Jonas Kastberg Hinrichsen, Léo Stefanesco, Lars Birkedal, and Amin Timany. Verifying Liveness Properties of Distributed Systems via Trace Refinement in Higher-Order Concurrent Separation Logic. 2024. 21, 31, 54

[48] T. Hoare. An axiomatic basis for computer programming. *CACM*, October 1969. doi: 10.1145/357980.358001. 14

[49] Aquinas Hobor, Robert Dockins, and Andrew W. Appel. A theory of indirection via approximation. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 171–184. ACM, 2010. doi: 10.1145/1706299.1706322. 36, 37

[50] Jan Hoffmann, Michael Marmar, and Zhong Shao. Quantitative Reasoning for Proving Lock-Freedom. *2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*, June 2013. doi: 10.1109/LICS.2013.18. 10

[51] Eleftherios Ioannidis, Yannick Zakowski, Steve Zdancewic, and Sebastian Angel. Structural temporal logic for mechanized program verification, October 2024. 8

[52] B. Jacobs, D. Bosnacki, and R. Kuiper. Modular Termination Verification of Single-Threaded and Multithreaded Programs. *ACM Trans. Program. Lang. Syst.*, July 2018. doi: 10.1145/3210258. 8, 25, 38, 43, 109

[53] Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,*

*POPL 2011, Austin, TX, USA, January 26-28, 2011*, Popl '11, pages 271–282, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926417. 104

[54] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and java. *nasa formal methods*, April 2011. doi: 10.1007/978-3-642-20398-5_4. 25, 28, 36

[55] Jules Jacobs and Stephanie Balzer. Higher-Order Leak and Deadlock Free Locks. *Proc. ACM Program. Lang.*, January 2023. doi: 10.1145/3571229. 10

[56] Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. Connectivity graphs: A method for proving deadlock freedom based on separation logic. *Proc. ACM Program. Lang.*, January 2022. doi: 10.1145/3498662. 10

[57] Prasad Jayanti and Srdjan Petrovic. Logarithmic-time single deleter, multiple inserter wait-free queues and stacks. In Ramaswamy Ramanujam and Sandeep Sen, editors, *FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science, 25th International Conference, Hyderabad, India, December 15-18, 2005, Proceedings*, Lecture Notes in Computer Science, pages 408–419. Springer, 2005. doi: 10.1007/11590156_33. 81, 94, 95, 96, 104

[58] Jonas Braband Jensen and Lars Birkedal. Fictional separation logic. *Lecture Notes in Computer Science*, March 2012. doi: 10.1007/978-3-642-28869-2_19. 17

[59] Xiao Jia, Wei Li, and Viktor Vafeiadis. Proving Lock-Freedom Easily and Automatically. *CPP*, January 2015. doi: 10.1145/2676724.2693179. 10

[60] Cliff B. Jones. Specification and design of (parallel) programs. In R. E. A. Mason, editor, *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, pages 321–332. North-Holland/IFIP, 1983. 36

[61] Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, October 1983. ISSN 0164-0925. doi: 10.1145/69575.69577. 106

[62] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris. May 2015. doi: 10.1145/2775051.2676980. 11, 17, 36, 37, 62, 65, 68, 80, 85, 104

[63] Ralf Jung, R. Krebbers, Jacques-Henri Jourdan, A. Bizjak, L. Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, January 2018. doi: 10.1017/S0956796818000151. 18, 19, 36, 53, 86, 101

[64] Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. The future is ours: Prophecy variables in separation logic. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. doi: 10.1145/3371113. 105

[65] Jieung Kim, Vilhelm Sjöberg, Ronghui Gu, and Zhong Shao. Safety and Liveness of MCS Lock—Layer by Layer. *Lecture Notes in Computer Science*, November 2017. doi: 10.1007/978-3-319-71237-6_14. 8, 64

[66] Naoki Kobayashi. Type Systems for Concurrent Processes: From Deadlock-Freedom to Livelock-Freedom, Time-Boundedness. *Lecture Notes in Computer Science*, August 2000. doi: 10.1007/3-540-44929-9_27. 10

[67] Naoki Kobayashi. Type Systems for Concurrent Programs. *Lecture Notes in Computer Science*, January 2003. doi: 10.1007/978-3-540-40007-3_26. 7, 10

[68] Naoki Kobayashi. A new type system for deadlock-free processes. In Christel Baier and Holger Hermanns, editors, *CONCUR 2006 - Concurrency Theory, 17th International Conference, CONCUR 2006, Bonn, Germany, August 27-30, 2006, Proceedings*, volume 4137 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2006. doi: 10.1007/11817949_16. 38

[69] Alex Kogan and Erez Petrank. Wait-free queues with multiple enqueuers and dequeuers. In Calin Cascaval and Pen-Chung Yew, editors, *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011*, SIGPLAN Not., pages 223–234. ACM, 2011. doi: 10.1145/1941553.1941585. 79, 107

[70] Alex Kogan and Erez Petrank. A methodology for creating fast wait-free data structures. In J. Ramanujam and P. Sadayappan, editors, *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012*, SIGPLAN Not., pages 141–150. ACM, 2012. doi: 10.1145/2145816.2145835. 94, 104, 107

[71] R. Krebbers, A. Timany, and L. Birkedal. Interactive proofs in higher-order concurrent separation logic. *POPL*, January 2017. doi: 10.1145/3009837.3009855. 11, 18

[72] Andrey Kupriyanov and Bernd Finkbeiner. Causal Termination of Multi-threaded Programs. *Lecture Notes in Computer Science*, July 2014. doi: 10.1007/978-3-319-08867-9_54. 7

[73] Pierre LaBorde, Steven D. Feldman, and Damian Dechev. A wait-free hash map. *International Journal of Parallel Programming*, 45(3):421–448, June 2017. ISSN 0885-7458. doi: 10.1007/s10766-015-0376-3. 92

[74] Ori Lahav, Egor Namakonov, Jonas Oberhauser, Anton Podkopaev, and Viktor Vafeiadis. Making weak memory models fair. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–27, October 2021. ISSN 2475-1421. doi: 10.1145/3485475. 3, 30

[75] Dongjae Lee, Minki Cho, Jinwoo Kim, Soonwon Moon, Youngju Song, and Chung-Kil Hur. Fair Operational Semantics. *Proc. ACM Program. Lang.*, June 2023. doi: 10.1145/3591253. 8, 36, 63

[76] Dongjae Lee, Janggun Lee, Tae-Hoon Yoon, Minki Cho, Jeehoon Kang, and Chung-Kil Hur. Lilo: A Higher-Order, Relational Concurrent Separation Logic for Liveness. *Proceedings of the ACM on Programming Languages*, April 2025. doi: 10.1145/3720525. 8, 9, 23, 24, 25, 38, 62, 63

[77] K. Rustan M. Leino, Peter Müller, and Jan Smans. Deadlock-free channels and locks. In Andrew D. Gordon, editor, *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*, volume 6012 of *Lecture Notes in Computer Science*, pages 407–426. Springer, 2010. doi: 10.1007/978-3-642-11957-6_22. 38

[78] N.G. Leveson and C.S. Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, July 1993. ISSN 0018-9162. doi: 10.1109/MC.1993.274940. 3

[79] Zhaohui Li and Xiaobing Feng. A program logic for obstruction-freedom. *Frontiers of Computer Science*, December 2023. doi: 10.1007/S11704-023-2774-9. 10

[80] Hongjin Liang and Xinyu Feng. A program logic for concurrent objects under fair scheduling. *POPL*, January 2016. doi: 10.1145/2837614.2837635. 9, 10, 36, 39, 43, 64, 79, 80, 106, 108

[81] Hongjin Liang, Xinyu Feng, and Zhong Shao. Compositional verification of termination-preserving refinement of concurrent programs. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, Csl-Lics '14, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 978-1-4503-2886-9. doi: 10.1145/2603088.2603123. 106

[82] Zongyuan Liu, Sergei Stepanenko, Jean Pichon-Pharabod, Amin Timany, Aslan Askarov, and Lars Birkedal. VMSL: A separation logic for mechanised robust safety of virtual machines communicating above FF-a. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023. doi: 10.1145/3591279. 54

[83] Raz Lotan and Sharon Shoham. Implicit Rankings for Verifying Liveness Properties in First-Order Logic, December 2024. 7

[84] Zohar Manna and Amir Pnueli. Completing the Temporal Picture,. December 1989. doi: 10.21236/ADA328579. 7

[85] Yusuke Matsushita and Takeshi Tsukada. Nola: Later-Free Ghost State for Verifying Termination in Iris. *Proceedings of the ACM on Programming Languages*, June 2025. doi: 10.1145/3729250. 8, 65

[86] Kenneth L. McMillan. Toward Liveness Proofs at Scale. *Lecture notes in computer science*, January 2024. doi: 10.1007/978-3-031-65627-9_13. 7

[87] Jia Meng, Lawrence C. Paulson, and Gerwin Klein. A termination checker for Isabelle Hoare logic. July 2007. 7

[88] Glen Mével, Jacques-Henri Jourdan, and François Pottier. Time Credits and Time Receipts in Iris. *european symposium on programming*, April 2019. doi: 10.1007/978-3-030-17184-1_1. 8

[89] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In James E. Burns and Yoram Moses, editors, *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, USA, May 23-26, 1996*, Podc '96, pages 267–275, New York, NY, USA, 1996. ACM. ISBN 0-89791-800-2. doi: 10.1145/248052.248106. 79

[90] I. N. Mulder. *Proof Automation for Fine-Grained Concurrent Separation Logic*. S.l. : s.n., 2025. ISBN 978-94-6496-310-6. 11

[91] Egor Namakonov, Lars Birkedal, and Amin Timany. Verifying wait-freedom for concurrent higher-order programs (Technical appendix), April 2026. 49, 99, 102

[92] Egor Namakonov, Lars Birkedal, and Amin Timany. Verifying Wait-Freedom for Concurrent Higher-Order Programs. *LIPIcs, Volume 372, ECOOP 2026*, 372:20:1–20:29, 2026. ISSN 1868-8969. doi: 10.4230/LIPICS.ECOOP.2026.20. 27, 28, 29

[93] Egor Namakonov, Lars Birkedal, and Amin Timany. Verifying wait-freedom for concurrent higher-order programs (Artifact), April 2026. 29, 102

[94] Egor Namakonov, Justus Fasse, Bart Jacobs, Lars Birkedal, and Amin Timany. Lawyer: Modular Obligations-Based Liveness Reasoning in Higher-Order Impredicative Concurrent Separation Logic. *Proceedings of the ACM on Programming Languages*, 10(OOPSLA1):132:1099–132:1125, April 2026. doi: 10.1145/3798240. 27, 29, 80, 85, 90, 91, 103, 104, 109

[95] Egor Namakonov, Justus Fasse, Bart Jacobs, Lars Birkedal, and Amin Timany. Artifact for the "lawyer: Modular obligations- based liveness reasoning in higher-order impredicative concurrent separation logic" paper of OOPSLA'26, February 2026. 29, 38, 39, 45, 50, 56, 61, 62, 63, 65, 66, 68, 74

[96] Susan Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of The Acm*, 19(5):279–285, May 1976. ISSN 0001-0782. doi: 10.1145/360051.360224. 37

[97] L. Padovani. Deadlock and lock freedom in the linear $\pi$-calculus. *CSL-LICS*, July 2014. doi: 10.1145/2603088.2603116. 10

[98] Christine Paulin-Mohring. Inductive Definitions in the system Coq - Rules and Properties. *international conference on typed lambda calculi and applications*, March 1993. doi: 10.1007/BFB0037116. 18

[99] Christina Peterson, Victor Cook, and Damian Dechev. Practical Progress Verification of Descriptor-Based Non-Blocking Data Structures. October 2019. doi: 10.1109/MASCOTS.2019.00019. 10, 80, 107

[100] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. TaDA: A Logic for Time and Data Abstraction. *Lecture Notes in Computer Science*, August 2014. doi: 10.1007/978-3-662-44202-9_9. 17, 25, 62, 104, 106

[101] Pedro da Rocha Pinto, Thomas Dinsdale-Young, Philippa Gardner, and Julian Sutherland. Modular Termination Verification for Non-blocking Concurrency. *Lecture Notes in Computer Science*, April 2016. doi: 10.1007/978-3-662-49498-1_8. 10, 79, 80, 105

[102] Andreas Podelski and Andrey Rybalchenko. A Complete Method for the Synthesis of Linear Ranking Functions. *Lecture Notes in Computer Science*, January 2004. doi: 10.1007/978-3-540-24622-0_20. 7

[103] Tobias Reinhard and Bart Jacobs. Ghost signals: Verifying termination of busy-waiting (technical report), 2021. 38, 43, 46

[104] Tobias Reinhard and Bart Jacobs. Ghost Signals: Verifying Termination of Busy Waiting. *Lecture Notes in Computer Science*, July 2021. doi: 10.1007/978-3-030-81688-9_2. 9, 23, 25, 26, 38, 43, 46

[105] Tobias Reinhard, Amin Timany, and B. Jacobs. A separation logic to verify termination of busy-waiting for abrupt program exit, July 2020. 25

[106] Naïm Moussaoui Remil and Caterina Urban. Termination Resilience Static Analysis. In Yu-Fang Chen, Thomas Jensen, and Ondřej Lengál, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 16417, pages 212–236. Springer Nature Switzerland, Cham, 2026. ISBN 978-3-032-15699-0 978-3-032-15700-3. doi: 10.1007/978-3-032-15700-3_11. 9

[107] Silvain Rideau and G. Winskel. Concurrent Strategies. *2011 IEEE 26th Annual Symposium on Logic in Computer Science*, June 2011. doi: 10.1109/LICS.2011.13. 8

[108] Michael Sammler. Automated and foundational verification of low-level programs. January 2023. 11

[109] Gerhard Schellhorn, Oleg Travkin, and Heike Wehrheim. Towards a Thread-Local Proof Technique for Starvation Freedom. *Lecture Notes in Computer Science*, June 2016. doi: 10.1007/978-3-319-33693-0_13. 10

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

[111] Tsubasa Shoshi, Takuma Ishikawa, Naoki Kobayashi, Ken Sakayori, Ryosuke Sato, and Takeshi Tsukada. Termination Analysis for the \(\pi \)-Calculus by Reduction to Sequential Program Termination, September 2021. 7

[112] Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. Transfinite Iris: Resolving an existential dilemma of step-indexed separation logic. *ACM-SIGPLAN Symposium on Programming Language Design and Implementation*, June 2021. doi: 10.1145/3453483.3454031. 8, 9, 20, 36, 65, 90

[113] Simon Spies, Lennard Gäher, Joseph Tassarotti, Ralf Jung, Robbert Krebbers, L. Birkedal, and Derek Dreyer. Later credits: Resourceful reasoning for the later modality. *Proceedings of the ACM on Programming Languages*, August 2022. doi: 10.1145/3547631. 19

[114] Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, Peter Schneider-Kamp, and Cornelius Aschermann. Automatically Proving Termination and Memory Safety for Programs with Pointer Arithmetic. *Journal of Automated Reasoning*, January 2017. doi: 10.1007/S10817-016-9389-X. 7

[115] Kasper Svendsen and Lars Birkedal. Impredicative Concurrent Abstract Predicates. *Lecture Notes in Computer Science*, April 2014. doi: 10.1007/978-3-642-54833-8_9. 19, 26, 63

[116] David Swasey, Deepak Garg, and Derek Dreyer. Robust and compositional verification of object capability patterns. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017. doi: 10.1145/3133913. 81, 100

[117] Joseph Tassarotti, Ralf Jung, and Robert Harper. A Higher-Order Logic for Concurrent Termination-Preserving Refinement, April 2017. 8, 65

[118] Amin Timany, Simon Oddershede Gregersen, Léo Stefanesco, Jonas Kastberg Hinrichsen, Léon Gondelman, Abel Nieto, and Lars Birkedal. Trillium: Higher-Order Concurrent and Distributed Separation Logic for Intensional Refinement. *Proceedings of the ACM on Programming Languages*, 8(POPL):241–272, January 2024. ISSN 2475-1421. doi: 10.1145/3632851. 9, 10, 20, 21, 22, 27, 36, 37, 38, 49, 51, 64, 66, 80, 85, 99, 105

[119] Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. Wait-free linked-lists. In J. Ramanujam and P. Sadayappan, editors, *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012*, pages 309–310. ACM, 2012. doi: 10.1145/2145816.2145869. 92, 94, 104

[120] Bogdan Tofan, Simon Bäumler, Gerhard Schellhorn, and Wolfgang Reif. Temporal logic verification of lock-freedom. *Lecture Notes in Computer Science*, June 2010. doi: 10.1007/978-3-642-13321-3_21. 10

[121] Caterina Urban, Arie Gurfinkel, and Temesghen Kahsai. Synthesizing Ranking Functions from Bits and Pieces. *Lecture Notes in Computer Science*, April 2016. doi: 10.1007/978-3-662-49674-9_4. 7

[122] Viktor Vafeiadis. *Modular Fine-Grained Concurrency Verification*. PhD thesis, University of Cambridge, Computer Laboratory / University of Cambridge, UK, July 2008. 104

[123] Niki Vazou. Liquid Haskell: Haskell as a Theorem Prover. January 2016. 7

[124] Martin T. Vechev and Eran Yahav. Deriving linearizable fine-grained concurrent objects. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 125–135. ACM, 2008. doi: 10.1145/1375581.1375598. 107

[125] Nicolas Voirol, Ravichandhran Madhavan, and Viktor Kuncak. Termination of Open Higher-Order Programs. January 2017. 7

[126] H. Xi. Dependent Types for Program Termination Verification. *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*, June 2001. doi: 10.1109/LICS.2001.932500. 7

[127] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees: Representing recursive and impure programs in Coq, December 2019. 8