Lilo: A Higher-Order, Relational Concurrent Separation Logic for Liveness

DONGJAE LEE, MIT, USA and Seoul National University, Republic of Korea
JANGGUN LEE, KAIST, Republic of Korea
TAEYOUNG YOON, Seoul National University, Republic of Korea
MINKI CHO, Seoul National University, Republic of Korea and FuriosaAI, Republic of Korea
JEEHOON KANG, KAIST, Republic of Korea
CHUNG-KIL HUR, Seoul National University, Republic of Korea

Concurrent separation logic (CSL) has excelled in verifying safety properties across various applications, yet its application to liveness properties remains limited. While existing approaches like TaDA Live and Fair Operational Semantics (FOS) have made significant strides, they still face limitations. TaDA Live struggles to verify certain classes of programs, particularly concurrent objects with non-local linearization points, and lacks support for general liveness properties such as "good things happen infinitely often". On the other hand, FOS's scalability is hindered by the absence of thread modular reasoning principles and modular specifications.

This paper introduces Lilo, a higher-order, relational CSL designed to overcome these limitations. Our core observation is that FOS helps us to maintain simple primitives for our logic, which enable us to explore design space with fewer restrictions. As a result, Lilo adapts various successful techniques from literature. It supports reasoning about non-terminating programs by supporting refinement proofs, and also provides Iris-style invariants and modular specifications to facilitate modular verification. To support higher-order reasoning without relying on step-indexing, we develop a technique called stratified propositions inspired by Nola. In particular, we develop novel abstractions for liveness reasoning that bring these techniques together in a uniform way. We show Lilo's scalability through case studies, including the first termination-guaranteeing modular verification of the elimination stack. Lilo and examples in this paper are mechanized in Coq.

CCS Concepts: • **Theory of computation** → **Separation logic**; **Logic and verification**; **Concurrency**.

Additional Key Words and Phrases: Liveness, fine-grained concurrency, higher-order logic, Coq

ACM Reference Format:
Dongjae Lee, Janggun Lee, Taeyoung Yoon, Minki Cho, Jeehoon Kang, and Chung-Kil Hur. 2025. Lilo: A Higher-Order, Relational Concurrent Separation Logic for Liveness. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 125 (April 2025), 29 pages. https://doi.org/10.1145/3720525

1 Introduction

Safety properties ("bad things never happen") and liveness properties ("good things eventually happen") have long been the standard for formulating program properties. Recently, *concurrent separation logic* (CSL) has excelled in verifying safety properties, enabling powerful modular reasoning across applications like Rust type systems [20], relaxed memory concurrency [9, 35], and distributed systems [27, 38].

Authors' Contact Information: Dongjae Lee, MIT, Cambridge, USA and Seoul National University, Seoul, Republic of Korea, dongjael@mit.edu; Janggun Lee, KAIST, Daejeon, Republic of Korea, janggun.lee@kaist.ac.kr; Taeyoung Yoon, Seoul National University, Seoul, Republic of Korea, taeyoung.yoon@sf.snu.ac.kr; Minki Cho, Seoul National University, Seoul, Republic of Korea and FuriosaAI, Seoul, Republic of Korea, minki.cho@furiosa.ai; Jeehoon Kang, KAIST, Daejeon, Republic of Korea, jeehoon.kang@kaist.ac.kr; Chung-Kil Hur, Seoul National University, Seoul, Republic of Korea, gil.hur@sf.snu.ac.kr.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2025 Copyright held by the owner/author(s).
ACM 2475-1421/2025/4-ART125
https://doi.org/10.1145/3720525

In contrast, CSL's application to liveness properties has been less extensive with very few existing work compared to safety properties. Among them, LiLi [30, 31] and TaDA Live [10] are two most notable approaches: LiLi focuses on the verification of progress properties of concurrent objects, and TaDA Live develops abstract reasoning principles for proving termination of concurrent programs. In particular, D'Osualdo et al. [10] observe that liveness reasoning can be carried out with an abstract notion of obligations that must be fulfilled, which enable thread-modular liveness reasoning.

However, both these state-of-the-art approaches have limitations. LiLi does not provide a modular specification that can be readily utilized in client proofs, meaning it (P1) lacks *intra-thread modularity*. On the other hand, TaDA Live's scope is also (P2) limited to termination verification, excluding general liveness properties like "good things happen infinitely often". Additionally, LiLi and TaDA Live both cannot verify concurrent objects with (P3) non-local linearization points (*e.g.,* helping). Moreover, (P4) none of LiLi and TaDA Live are *mechanized*.

We observe that the limitations of previous approaches comes from their *inflexible primitives* for liveness reasoning. As shown by Alpern and Schneider [1], proving a liveness property requires a well-foundedness argument. To establish such arguments, previous approaches have incorporated abstractions and complex proof rules as primitives that are strongly *tied* to the underlying semantics and utilized in the proof of soundness to construct well-foundedness arguments. This restriction hinders extending their approaches to more diverse classes of programs.

Based on this observation, we recognize that the recently proposed Fair Operational Semantics (FOS) [28] enable us to develop CSL for liveness with simple primitives. In particular, FOS develops a novel approach to *scheduler fairness* called the *fairness counter*. Scheduler fairness, which states that every thread is eventually scheduled, is crucial for verifying liveness properties in most concurrent programs, and the fairness counter helps us maintain simple yet sufficient primitives that enable us to reason about scheduler fairness. Lee et al. [28] also developed a CSL for proving refinements between concurrent programs, but as we show in §2, their approach lacks scalability because (P5) it does not support *inter-/intra-thread modular* liveness reasoning.

Contributions. We develop Lilo, a higher-order, relational CSL for liveness with *scalable reasoning principles*, applicable to *broader classes of programs* and fully *mechanized* in Coq. Our reasoning principle is based on the following observation: to ensure liveness, the "good thing" must happen within some *finite number of program steps*. We implement this principle on top of FOS to leverage the power of the fairness counters and maintain simple primitives for our logic. In summary:

(1) Lilo facilitates inter-/intra-thread modular liveness reasoning through abstractions called *progress credits*, *obligation list*, and *promises* (addresses (P5)).
(2) Lilo extends beyond termination to prove general liveness properties, including those involving non-termination, through refinement proofs between concurrent programs (addresses (P2)).
(3) Lilo incorporates flexible Iris-style invariants [22, 24] and powerful modular specifications that support non-local linearization points (addresses (P1), (P3), (P5)).
(4) Lilo and verification examples are all mechanized in Coq [2] (addresses (P4)).

We note that Iris-style invariants require higher-order reasoning, for which a standard approach is step-indexing. However, step-indexing is ill-suited for liveness reasoning and requires extensions to support liveness, which is still an ongoing area of research [14, 40, 45]. We *do not* aim to extend step-indexing to support liveness. Instead, we develop a workaround called *stratified propositions* inspired by Nola [32]. This enables higher-order liveness reasoning *without* step-indexing.

We show Lilo's scalability via several case studies. In particular, Iris-style safety proofs for concurrent data structures' functional correctness extend naturally to functional correctness *and*

termination, as shown in our verification of concurrent stacks. Notably, to the best of our knowledge, we prove the first termination-guaranteeing modular specification for the elimination stack [15].

Outline. §2 provides background on TaDA Live and FOS, motivating Lilo. §3 introduces Lilo's core abstractions for reasoning about liveness guarantees made by scheduler fairness. §4 demonstrates how Lilo enables termination-guaranteeing modular specifications. §5 introduces *stratified propositions*, a technique that enables higher-order reasoning without step-indexing. §6 discusses Lilo's generalized rules. §7 demonstrates Lilo's proof scalability through case studies. §8 concludes with related work. Supplementary material [2] provides Coq mechanization of Lilo and examples.

2 Background and Motivation

Lilo is motivated by the abstractions in previous concurrent separation logics (CSLs) for liveness (§2.1), and built upon Fair Operational Semantics (§2.2, §2.3).

2.1 Concurrent Separation Logics for Liveness

There exists few CSLs for liveness, and LiLi [30, 31] and TaDA Live [10] are the two most notable approaches. LiLi focuses on the verification of progress properties of concurrent objects by establishing contextual refinement between two programs, but does not provide modular specifications (specs) that enable intra-thread modular verification. On the other hand, TaDA Live focuses on developing abstract reasoning principles and modular termination-guaranteeing specs for library functions, based on the abstract notion of obligations to be fulfilled.

However, previous approaches currently fall short in verifying important classes of programs.

Non-local linearization point. Both LiLi and TaDA Live does not support *non-local linearization point*, a pattern that employs inter-thread cooperation (such as helping) and is widely used in concurrent data structures such as the ticket lock and the elimination stack (P3). Reasoning about non-local linearization points requires careful design of logical primitives. For example, TaDA Live is based on TaDA, and a discussion by da Rocha Pinto [7, §8] shows that extending TaDA to support helping requires significant changes to the logic because TaDA's semantics ties the notion of atomic update to a concrete atomic operation in the implementation. Although supporting non-local linearization points is considered orthogonal to liveness reasoning in previous works [10, 30, 31], designing a logic that handles both non-local linearization points and liveness is an open question.

Non-termination. TaDA Live cannot reason about (possibly) *non-terminating* programs (P2). This limitation is significant because non-terminating programs pose unique challenges to the liveness reasoning. For instance, consider the following "infinite message passing" example:

$$\begin{array}{l}\texttt{while}\ (1)\ \{\ X \coloneqq 1;\\ \quad \texttt{do}\ \{\ a \coloneqq X;\ \}\ \texttt{while}\ (a = 1);\ \mathit{print}(a);\ \}\end{array} \ \Big\|\ \begin{array}{l}\texttt{while}\ (1)\ \{\ X \coloneqq 2;\\ \quad \texttt{do}\ \{\ b \coloneqq X;\ \}\ \texttt{while}\ (b = 2);\ \mathit{print}(b);\ \}\end{array} \qquad \text{(INF-MP)}$$

This program has two threads, where the first and second threads pass messages to each other by respectively writing 1 and 2 to the shared location X, and then wait for the other thread to update X in the inner loops. When each thread receives the update, it exits the inner loop and prints the updated value, which is 2 for the first thread and 1 for the second thread. Moreover, this message passing is repeated indefinitely by the outer while loop, therefore called "infinite message passing".

Under scheduler fairness, both threads eventually make progress (*i.e.,* print and start a new iteration). This is because whenever a thread updates X, the other thread eventually reads the updated value, exits the inner loop, prints, and then starts a new iteration in which it updates X. This scenario requires reasoning about indefinite dependencies between liveness assumptions ("will update X"), and it is unclear how to reason about this with TaDA Live.

Scheduler non-determinism. LiLi and TaDA Live cannot verify programs that require direct reasoning about *scheduler non-determinism*. For example, consider the following example:

$$\texttt{while}\ (d = 0)\ \{\ \texttt{lock}(x);\ d \coloneqq D;\ \texttt{unlock}(x);\ \}\ \Big\|\ D \coloneqq 1; \qquad \text{(SCH-ND)}$$

which is a simplified version of a program by D'Osualdo et al. [10, §5.6]. The first thread waits for the second thread to update the shared memory location D in a loop, and it acquires and releases a spinlock during each iteration. This program is guaranteed to terminate under scheduler fairness because the second thread will eventually be executed and write 1 to D. Consequently, the first thread will eventually read the updated value from D and exit the loop.

However, one cannot carry out this reasoning with the TaDA Live spec of spinlock, which requires the client to specify an upper bound on the number of calls to the lock function before the second thread gets scheduled. This bound is determined solely by the scheduler, but TaDA Live does not provide any means to access this information. LiLi also suffers the same limitation, because it cannot directly reason about scheduler non-determinism.

How Lilo tackles problems of the previous CSLs. Our core observation is that logical primitives of LiLi and TaDA Live are not flexible enough to reason about aforementioned classes of programs. Lilo has simple primitives, *i.e.,* the model of separation logic (such as PCMs), the simulation relation, and the fairness counter inherited from FOS, and all abstractions and rules are developed on top of these primitives. This simple set of primitives allow the development of abstractions and proof rules with fewer restrictions, *i.e.,* separate them from the underlying semantics.

As a result, Lilo adapts Iris's well-known safety reasoning for "non-local linearization point" to liveness setting, and we verify a ticket lock in §7.1 and the elimination stack in §7.2. At the same time, Lilo can handle "non-termination" and "scheduler non-determinism" by leveraging the refinement-based relational reasoning of FOS, and we verify INF-MP and SCH-ND in §7.3.

2.2 Overview of Fair Operational Semantics

Fair Operational Semantics (FOS) [28] is a style of operational semantics that can express various notions of fairness. Lilo is built upon FOS, utilizing its programming language, refinement, and simulation relation. We provide an overview of FOS that can help readers to follow the rest of the paper, and refer the interested readers to Lee et al. [28] for further detail, in particular §4 and §5.

Language with fairness events. The programming language of FOS supports non-determinism, modeled with the $\texttt{PICK}(X)$ instruction that non-deterministically returns a value from the set X, and *fairness events* $f \in \{\text{good}, \text{bad}\}$, invoked by the $\texttt{FAIR}(f)$ instruction. Intuitively, fairness events define the set of fair execution traces of a program by allowing only *finite* bads to be invoked until a good is invoked. Although fairness events can express various notions of fairness, Lilo only utilizes scheduler fairness. Specifically, fair schedulers can be modeled by invoking a fairness event for each thread whenever a thread is scheduled. When a thread is scheduled, by invoking a good for the scheduled thread and a bad for the remaining threads, every thread is guaranteed to be eventually scheduled because bad cannot continue indefinitely without a good.

Moreover, FOS models concurrency by interleaving semantics where the scheduler picks a thread to execute and a thread yields to the scheduler. Then, scheduler fairness in FOS is defined as an abstract program that utilizes the FAIR constructor to express fairness. In addition, the programming language has the yield constructor $\mathcal{Y}$ that a thread can execute to yield to the scheduler, which is commonly used to make the interleaving explicit [11, 25].

Refinement and simulation relation. In FOS, an implementation program (*target*) *refines* a specification program (*source*) if the behavior of the target is included in that of the source. The behavior of a program is the set of observable traces of the program, which includes termination

and non-termination (therefore the refinement is termination-preserving). What makes FOS special is the fact that the fairness events invoked by FAIR defines *fair* traces, enabling *fair* refinement that compares only fair behaviors of two programs.

To prove fair refinements, FOS develops a *thread-local* simulation relation and a technique called fairness counter. A fairness counter intuitively corresponds to the number of bads that can be invoked before a good. In the case of scheduler fairness, each thread has its own fairness counter that decreases when the corresponding thread is not scheduled and resets to an arbitrary number when it is scheduled. The thread-local simulation relation utilizes the fairness counter to reason about the FAIR constructor. For further details, please refer to Lee et al. [28, §2].

2.3 Liveness Reasoning in FOS

Lee et al. [28] also develop Fairness Logic, a thread-local program logic designed to prove fair refinements between source and target programs. Fairness Logic is a relational separation logic based on *simulation weakest precondition* [13], which comes with two sets of proof rules, each for reasoning about the source or the target program.

On the one hand, Fairness Logic's source-side liveness reasoning is simple thanks to the powerful simulation rules that abstract away the complexities of scheduler fairness in the source. Moreover, source programs in refinement proofs are generally much simpler compared to target programs, which makes source side reasoning straightforward. Therefore, Lilo directly imports these rules.

On the other hand, it is challenging to scale Fairness Logic's target-side reasoning due to the lack of the following three principles (P5).

Inter-thread modularity. Fairness Logic's target-side reasoning principle for liveness is not modular for multiple threads. In FOS, scheduler fairness can be harnessed through a globally shared ghost state called *fairness counter* that assigns a natural number to each thread. The natural number decreases whenever the associated thread is not scheduled, and scheduler fairness guarantees that every thread is scheduled before its fairness counter runs out. Fairness Logic develops an abstraction of fairness counter in the form of separation logic assertions called *fairness assertions*, which represent *thread-local views* of fairness counter. However, despite the fact that fairness assertions represent thread-local views, liveness reasoning in Fairness Logic breaks this locality by requiring fairness assertions to be *shared* with other threads.

For example, consider the following example from Lee et al. [28, §7.2] (slightly modified):

$$\texttt{skip}; X \coloneqq 1; \textbf{ret}\ 0 \ \Big\|\ \textbf{do}\ \{\ a \coloneqq X;\ \}\ \textbf{while}\ (a = 0); \textbf{ret}\ a \tag{MP}$$

MP terminates as the first thread will eventually be scheduled, process skip, write 1 to the shared location X, and terminate with a return value 0. Similarly, the second thread will eventually read this updated value, exit the loop, and then terminate with a return value a (which will be 1).

To prove the termination of MP using fairness assertions in Fairness Logic, one has to expose thread-local details to the shared invariant breaking thread-local modularity. Specifically, the shared proof invariant presented in Lee et al. [28, §7.2] includes a fairness assertion designated to the first thread, and also encodes the *number of remaining instructions* for the first thread to reach $X \coloneqq 1$.[1] Such a proof that exposes thread-local views and depends directly on low-level implementation details is fragile, and slight modifications to the program can easily break it.

Intra-thread modularity. Fairness Logic's target-side liveness rules are not modular within a single thread, making it difficult to capture liveness guarantees of library functions. In Fairness Logic, liveness reasoning for library functions depends on fairness assertions, which are often shared among threads as seen in MP. The ability to reason about such *shared assertions* is severely limited

[1] More precisely, it encodes the number of remaining $\mathcal{Y}$ (Yield) instructions, which will be explained in §3.

in Fairness Logic compared to existing (safety reasoning) logics such as TaDA [8] and Iris [24]. As a result, it is unclear how to develop modular specs (*e.g.,* Hoare triples) with Fairness Logic.

Indeed, instead of developing modular specs, Fairness Logic proves contextual refinement between a library module and a spec module and a client of the library needs to inline the spec code to carry out reasoning. For example, a spec for a spinlock in Fairness Logic is the code of the spinlock itself, which does not capture liveness properties such as termination conditions of a spinlock.

Nested structure of liveness reasoning. The main complexity in liveness reasoning often rises from *nested structures* of liveness arguments, and as pointed out by Lee et al. [28, §9], Fairness Logic does not provide any abstraction for expressing and reasoning about such structure. For example, to prove the termination of MP using Fairness Logic, we need to manually construct a logical assertion representing a monotonically decreasing tuple (l, n), where l represents the number of remaining instructions for the first thread to execute $X \coloneqq 1$ and n represents the value of the fairness counter of the first thread. The degree of low-level effort required for the user hinders scaling Fairness Logic to examples involving more complicated liveness arguments.

How Lilo tackles problems of FOS. Lilo enables scalable "inter-thread modularity" through abstractions called *obligation list and promise*, which is inspired by TaDA Live, as we show in §3 by proving MP. In addition, Lilo supports flexible "intra-thread modularity" through *higher-order reasoning* such as Iris style invariants, and we develop library specs that guarantee functional correctness *and* termination in §4 and §7. Also, Lilo facilitates proofs involving "nested structure of liveness reasoning" by developing a technique called *obligation link*, as we demonstrate in §4.

3 Liveness Reasoning under Scheduler Fairness

In this section, we introduce the core abstractions of Lilo that enable *thread-modular reasoning* about *liveness guarantees made by scheduler fairness*. We demonstrate the abstractions and rules of Lilo by proving the termination of the MP program presented in §2.3 as a running example.

Scheduler fairness. We first elaborate the example with *yield* instructions [28, §5.1]:

$$\mathcal{Y};\ \texttt{skip};\ \mathcal{Y};\ X \coloneqq 1;\ \mathcal{Y};\ \texttt{ret}\ 0 \ \Big\|\ \texttt{do}\ \{\ \mathcal{Y};\ a \coloneqq X;\ \mathcal{Y};\ \}\ \texttt{while}\ (a = 0);\ \mathcal{Y};\ \texttt{ret}\ a \tag{MP}$$

The yield instruction $\mathcal{Y}$ is a vital primitive in FOS that enables the *interleaving semantics* of concurrency: whenever a thread executes $\mathcal{Y}$, it yields to the scheduler, which then decides the next thread to execute. In contrast, a sequence of instructions without yields is executed atomically. Moreover, the yield-based interleaving semantics in FOS enables a natural description of *scheduler fairness*, which means that the scheduler guarantees that *every thread will eventually be scheduled*.

As we discussed in §2.3, MP terminates because the first thread (thread 1) *eventually* writes 1 to the shared location X. Thanks to scheduler fairness, it eventually gets scheduled, and every time it gets scheduled, it is closer to executing $X \coloneqq 1$. The second thread (thread 2) reads X in a loop. If it reads 1, it exits the loop and terminates with a return value of 1. If it reads 0, it knows that, thanks to scheduler fairness, thread 1 is one step closer to being scheduled, and thus, one step closer to writing 1 to X. Therefore, thread 2 cannot read 0 indefinitely, and eventually terminates.

Background: termination proof in FOS. In FOS, we carry out this termination proof by defining a source program and proving thread-local simulations using fairness assertions. For example,

$$\mathcal{Y};\ \texttt{ret}\ 0 \ \Big\|\ \mathcal{Y};\ \texttt{ret}\ 1\,, \tag{$\mathrm{MP_S}$}$$

is a possible source program for our proof, where the two threads immediately terminate after a yield, returning values 0 and 1 respectively. Then, the simulation proof implies that MP refines $\mathrm{MP_S}$, and by the definition of *termination-sensitive* refinement, implies the termination of MP. As mentioned in §2.3, FOS provides proof rules in the form of *simulation weakest precondition* [13],

$$\textsc{cred-new}\ \frac{\ell,\ n \in \mathbb{N}}{\dot{\Rightarrow} \exists \kappa,\ \blacklozenge_\kappa \lceil \ell,\ n \rceil * \diamondsuit_\kappa(\ell,\ n)} \qquad \textsc{pc-split}\ \frac{\diamondsuit_\kappa(\ell,\ n_1 + n_2)}{\diamondsuit_\kappa(\ell,\ n_1) * \diamondsuit_\kappa(\ell,\ n_2)} \qquad \textsc{pc-drop}\ \frac{\diamondsuit_\kappa(\ell_2,\ 1) \quad \ell_1 < \ell_2 \quad n \in \mathbb{N}}{\dot{\Rightarrow} \diamondsuit_\kappa(\ell_1,\ n)}$$

$$\textsc{obls-add}\ \frac{\mathit{Obls}_{th}(\Phi) * \diamondsuit_\kappa(1,\ 1) \quad \mathsf{persistent}(P)}{\dot{\Rightarrow} \mathit{Obls}_{th}((\kappa, P) :: \Phi)} \qquad \textsc{obls-fulfill}\ \frac{\mathit{Obls}_{th}(\Phi) * P \quad (\kappa, P) \in \Phi}{\dot{\Rightarrow} \mathit{Obls}_{th}(\Phi \setminus (\kappa, P))} \qquad \textsc{prom-get}\ \frac{\mathit{Obls}_{th}(\Phi) \quad (\kappa, P) \in \Phi}{\xrightarrow{\kappa} \diamond P}$$

Fig. 1. Selected and simplified rules of Lilo related to liveness obligations.

which are sufficient for the source-side liveness reasoning. Therefore, *we directly import the source side proof rules of FOS into Lilo*. On the other hand, proof rules of FOS for the target side liveness reasoning fail to support thread-modular liveness reasoning, making it difficult to scale.

Termination proof in Lilo. Lilo aims to improve target side liveness reasoning in FOS by developing abstractions on top of FOS, based on the following observation: to ensure liveness, the "good thing" must happen within some *finite number of program steps*.

Specifically, in Lilo, thread 1 is assigned an *obligation* to write 1 to X, and also given a finite amount of *credits* that can only monotonically decrease. With this obligation, thread 1 must *submit* certain amount of credits whenever it executes a yield, where executing a yield corresponds to taking a program step in Lilo, until it fulfills the obligation by writing 1 to X. Because thread 1 has a finite amount of credits, it can execute only a finite number of yields before fulfilling its obligation. Therefore, we can ensure that "writing 1 to X" happens within some bounded number of thread 1's execution of yields, where the bound is decided by the initial amount of the credits.

Thread 1 can also let other threads know that it currently holds an obligation to write 1 to X. Then, thread 2 knows that thread 1 is scheduled due to scheduler fairness, and *also* knows that thread 1 is getting closer to writing 1 to X due to the obligation assigned to the first thread. Based on this argument, thread 2 knows it will eventually read 1 from X, exit the loop, and terminate. Lilo's abstractions and proof rules capture this argument, enabling thread-modular liveness reasoning.

Formally, we prove the termination of MP by proving refinement between MP and MP_S through a powerful simulation weakest precondition, equipped with flexible Iris style *invariants* [22] and rules for thread-modular liveness reasoning. Specifically, to establish refinement between two programs, we prove $tgt \lesssim^{th}_{\mathcal{E}} src$, which is an assertion denoting the weakest precondition for proving thread-local simulation between tgt (target) and src (source), under invariants represented by the *invariant mask* $\mathcal{E}$ and the parameterized thread id th.

For the rest of the section, we first show how Lilo ensures that thread 1 writes 1 to X using the notion of obligations. Then we show how Lilo allows thread 2 to eventually read 1 from X. Finally, we close the section with a discussion about the simulation rules of Lilo.

3.1 Ensuring Progress of the First Thread

As we discussed, we need to ensure that thread 1 eventually writes 1 to the shared memory location X in order to prove the termination of our running example MP. Lilo enables this reasoning with two separation logic predicates: *progress credits* $\diamondsuit_\kappa(\ell,\ n)$ and *obligation lists* $\mathit{Obls}_{th}(\Phi)$ (Fig. 1).

Progress credits. In Lilo, a liveness obligation, or simply an obligation, is just an identifier, usually denoted by κ. Every obligation κ comes with a *finite* amount of *progress credits*, and we use the CRED-NEW rule to obtain a new obligation κ and a desired amount $(\ell,\ n)$ of its progress credits $\diamondsuit_\kappa(\ell,\ n)$ (we introduce the other predicate $\blacklozenge_\kappa \lceil \ell,\ n \rceil$ later, in §3.2). When we have $\diamondsuit_\kappa(\ell,\ n)$, we say that we have n progress credits of *layer* ℓ for the obligation κ. For now, we can ignore the layer part and assume $\ell = 1$; we explain the layer part in the end of this subsection.

Once we obtain $\diamond_\kappa(\ell,\ n)$, we cannot increase the total amount of $\diamond_\kappa(\ell,\ n)$; *i.e.,* the total amount of progress credits for κ can only monotonically *decrease*. Additionally, Lilo has a "yield-tax" rule (explained formally in §3.3): if a thread has an obligation κ, it must submit $\diamond_\kappa(1,\ 1)$ whenever it executes a yield, until the thread fulfills κ. Because we only have a finite amount of $\diamond_\kappa(\ell,\ n)$ that we can spend, "yield-tax" ensures that a thread with an obligation κ can execute only a bounded number of yields, unless it fulfills κ.

Obligation list. We assign an obligation to a thread using an abstraction called *obligation list*, denoted Φ, which is a list of pairs $(\kappa,\ P)$ where κ is an obligation and P is a *sProp*.[2] An obligation κ is assigned to a thread *th* together with a proposition P that should be fulfilled by the thread, by adding the pair $(\kappa,\ P)$ to the obligation list Φ of thread *th*. Each thread owns a *private* obligation list, and the predicate $\mathit{Obls}_{th}(\Phi)$ means that the thread *th* currently owns an obligation list Φ. Because $\mathit{Obls}_{th}(\Phi)$ essentially represents Φ, we overload the term "obligation list" to refer to both of them.

Formally, we use OBLS-ADD to add an obligation κ to eventually establish some predicate P to a thread *th*'s obligation list. This rule requires progress credits $\diamond_\kappa(1,\ 1)$, for the same reason as "yield-tax" requires $\diamond_\kappa(1,\ 1)$; we explain this in §3.2. After thread *th* establishes P, it can remove κ from its obligation list using OBLS-FULFILL. We note that the propositions P added to obligation lists are *persistent*, meaning that the proposition can be duplicated, and once a thread establishes P, P remains true forever. However, this constraint is not a severe drawback because Lilo supports Iris style invariants that enable us to allocate a persistent invariant when needed.

In addition, a thread can share its obligation by sharing a persistent predicate $\xrightarrow{\kappa}\!\!\diamond P$ called a *promise*, that can be created using PROM-GET. Note that the premise is not consumed when a rule gives a persistent predicate, so $\mathit{Obls}_{th}(\Phi)$ is not consumed by PROM-GET. Then, a thread can create and share $\xrightarrow{\kappa}\!\!\diamond P$ with other threads to let them know that there is a thread with an obligation κ to establish P. Later, in §3.2, we show how other threads can rely on $\xrightarrow{\kappa}\!\!\diamond P$ to obtain P in their proofs.

Termination proof of thread 1. Proving the termination of thread 1 is straightforward because it does not contain any loops. However, we must establish the liveness argument that "eventually, the value of X is updated to 1", so that thread 2 can rely on this argument for termination. This argument is carried out by adding an obligation κ to thread 1's obligation list *as an initial condition*, and fulfilling it by writing 1 to X. We present a proof outline here, without formally introducing the simulation and invariant rules of Lilo, which we introduce in §3.3.

Concretely, we prove the following thread-local simulations for thread 1 (Sim1):

$$\mathit{Obls}_1([(\kappa,\ \boxed{X \mapsto 1}^{w})]) \ *\ \diamond_\kappa(1,\ 2) \ *\ \mathit{pend}_{1/2} \ *\ \boxed{\mathrm{MP_I}}^{\nu} \ \twoheadrightarrow\ \mathit{y};\ \mathtt{skip};\ \mathit{y};\ X \coloneqq 1;\ \mathit{y};\ \mathbf{ret}\ 0 \ \precsim^{1}_{\top}\ \mathit{y};\ \mathbf{ret}\ 0$$

Here, $\boxed{X \mapsto 1}^{w}$ and $\boxed{\mathrm{MP_I}}^{\nu}$ are *invariants*: $\boxed{I}^{\nu}$ is a persistent predicate that means we can *open* the invariant to obtain a proposition I, where ν is an identifier. However, we must *close* the invariant by proving I to execute a yield or a **ret**. For example, we can open $\boxed{X \mapsto 1}^{w}$ to obtain a points-to predicate $X \mapsto 1$ meaning that the value at X is 1. We explain invariants more in §3.3.

The proof invariant $\mathrm{MP_I}$ is defined as follows:

$$\mathrm{MP_I} \triangleq ((\mathit{pend}_{1/2} \ *\ X \mapsto 0) \ \vee\ (\mathit{shot} \ *\ \boxed{X \mapsto 1}^{w})) \tag{$\mathrm{MP_I}$}$$

where pend_q and *shot* are the usual oneshot assertions with the following rules ($0 < p,\ q \le 1$):

$$\mathit{pend}_p * \mathit{pend}_q \vdash \mathit{pend}_{p+q} \qquad \mathit{pend}_1 \vdash \Rightarrow \mathit{shot} \qquad \mathit{pend}_q * \mathit{shot} \vdash \mathsf{False} \qquad \mathsf{persistent}(\mathit{shot})$$

This invariant describes a state transition system with two states (a) $\mathit{pend}_{1/2} * X \mapsto 0$ and (b) $\mathit{shot} * \boxed{X \mapsto 1}^{w}$, and the oneshot assertions ensure that the transition is one-way, from (a) to (b).

[2]The type *sProp* is the type of propositions in Lilo. We abstract away the details of *sProp* until we introduce it in §5.

It is easy to check that the initial condition holds, given that the memory location X is initialized to 0, using rules such CRED-NEW and OBLS-ADD.

To prove termination, we need to execute the two yields and a skip before the memory write to X: $\mathcal{Y}$; skip; $\mathcal{Y}$; $X \coloneqq 1$; . Because thread 1 has an obligation κ, $\mathit{Obls}_1([(\kappa, \boxed{X \mapsto 1}^w)])$, by the "yield-tax" rule, we need to submit $\diamond_\kappa(1, 1)$ when we execute the two yields. We can apply the PC-SPLIT rule to $\diamond_\kappa(1, 2)$ in the initial condition and obtain two $\diamond_\kappa(1, 1)$s, and submit each of them when executing the two yields. Also, a skip can be executed in a trivial way.

After executing the two yields and a skip, we reach the memory write: $X \coloneqq 1$; $\mathcal{Y}$; **ret** 0. To obtain the points-to predicate, we open the invariant $\mathrm{MP_I}$ and obtain the disjunction. Using the rules of oneshot assertions together with $\mathit{pend}_{1/2}$ in the initial condition, we obtain the $(\mathit{pend}_{1/2} * X \mapsto 0)$ case and eliminate the other case of the disjunction. Then, we execute $X \coloneqq 1$; with $X \mapsto 0$ and obtain $X \mapsto 1$, and wrap it into a new invariant $\boxed{X \mapsto 1}^w$ using the invariant rules. At the same time, we combine the two $\mathit{pend}_{1/2}$s to obtain pend_1, and update it to *shot*. This allows us to close the invariant $\mathrm{MP_I}$ by proving the $(\mathit{shot} * \boxed{X \mapsto 1}^w)$ part of the disjunction.

In addition to this, thread 1 can *fulfill* its obligation κ because it established the promised proposition $\boxed{X \mapsto 1}^w$. Using the OBLS-FULFILL rule with $\mathit{Obls}_1([(\kappa, \boxed{X \mapsto 1}^w)])$, we can fulfill κ and obtain an empty obligation list $\mathit{Obls}_1([])$. Then, the rest of the proof for $\mathcal{Y}$; **ret** 0 is straightforward: because thread 1 has no obligation, it can execute the last yield without submitting any progress credits, and terminate by executing **ret** 0. We note that a thread must have an empty obligation list and close every invariant to execute **ret** for soundness.

Extending progress credits with a layer. As we discussed, Lilo's principle for liveness reasoning is to bound the number of yields a thread can execute when holding an obligation. If a thread has an obligation in its obligation list, it must submit a certain amount of progress credits to execute a yield, until the obligation is fulfilled. Therefore, it is crucial for the user to determine the right amount of progress credits when using CRED-NEW to get the credits. Then, we have an important question: *how do we choose the number of progress credits?* When the number of yields is deterministic, we can count it and prepare enough amount of progress credits. For the case of thread 1, we need $\diamond_\kappa(1, 2)$ to execute the two yields before the memory write.

However, choosing the right number of credits becomes tricky when programs involve *non-determinism*. Consider a modified MP, where PICK non-deterministically returns a natural number:

$$\mathcal{Y}; n \coloneqq \mathtt{PICK}(\mathbb{N}); \mathbf{do}\ \{\ \mathcal{Y}; n{-}{-}; \}\ \mathbf{while}\ (n > 0); \mathcal{Y}; X \coloneqq 1; \mathcal{Y}; \mathbf{ret}\ 0$$

Here, the number of yields we need to execute before $X \coloneqq 1$ is $n + 2$, where n is obtained only after executing $\mathtt{PICK}(\mathbb{N})$. Therefore, we do not know the number of progress credits we need when we use CRED-NEW to allocate progress credits for the initial condition.

To handle non-determinism, we extend the progress credit with the notion of *layer*, which intuitively corresponds to the depth of non-determinism. Given a progress credit $\diamond_\kappa(\ell, n)$, ℓ is the layer of this credit and the pair (ℓ, n) represents the amount of progress credits, following the *lexicographical order* on the product $\mathbb{N} \times \mathbb{N}_{>0}$ as demonstrated in PC-SPLIT and PC-DROP. Therefore, for our modified code, we can start the proof with $\diamond_\kappa(2, 2)$, obtain $\diamond_\kappa(1, 1)$ and $\diamond_\kappa(2, 1)$ using PC-SPLIT and PC-DROP, and execute the first yield using $\diamond_\kappa(1, 1)$. Then, we execute $n \coloneqq \mathtt{PICK}(\mathbb{N})$, determine n, and obtain $\diamond_\kappa(1, n + 1)$ by PC-DROP. This credit is enough to execute the remaining yields until $X \coloneqq 1$, and we can finish the proof for the modified code.

The notion of layer is crucial in Lilo because liveness reasoning for concurrent programs involves surprising amount of non-determinism. We discuss more instances in the rest of the paper.

PROM-PERS
$\mathsf{persistent}(\xrightarrow{\kappa}\!\!\diamond P)$

PROM-PROGRESS
$$\frac{€ * \xrightarrow{\kappa}\!\!\diamond P}{\Rightarrow\!\!\!\!\cdot\, \diamond_\kappa(0,\,1) \lor P}$$

CB-PERS
$\mathsf{persistent}(\blacklozenge_\kappa\lceil \ell,\, n\rceil)$

CRED-IND
$$\frac{\blacklozenge_\kappa\lceil \ell,\, n\rceil * \Box((\diamond_\kappa(0,\,1) \Rrightarrow\!\!\!\!\ast\; Q) \Rrightarrow\!\!\!\!\ast\; Q)}{\Rightarrow\!\!\!\!\cdot\, Q}$$

Fig. 2. Selected and simplified rules of Lilo related to induction.

3.2 Proving Termination of the Second Thread

The termination of thread 2 depends on the following argument: thread 1 will eventually get scheduled by scheduler fairness, and thread 1 will eventually write 1 to X. The first part is guaranteed by the underlying FOS, and the second part is ensured by the rules of progress credits and obligation lists. In this section, we show how we can carry out this argument in Lilo by introducing two core rules: PROM-PROGRESS and CRED-IND (Fig. 2).

Lilo's soundness and PROM-PROGRESS. We explain the PROM-PROGRESS rule in depth, because it is directly related to Lilo's soundness. To do so, we first need to explain *fairness counter* (Lee et al. [28, §2.2]), a logical state of FOS that represents scheduler fairness. Fairness counter **fc** is a map from thread ids to natural numbers, assigning a natural number to each active thread. Also, we have the "yield-fc" rule: *whenever a thread th is scheduled, its counter* $\mathbf{fc}(th)$ *is reset to an arbitrary number and the counter of other threads decrement.* FOS ensures that no thread's counter drops below 0, meaning that every thread is eventually scheduled, guaranteeing scheduler fairness.

Then, the *model* of obligation list connects liveness reasoning in Lilo to the fairness counter by maintaining the following (greatly simplified) global invariant:

$$\mathit{Obls}_{th}([(\kappa, P)]) \approx (\exists n.\, \mathbf{fc}(th) = n \,*\, \diamond_\kappa(0,\, n)) \lor P$$

This invariant is initially established internally through OBLS-ADD, which requires $\diamond_\kappa(1,\,1)$ because the value n of $\mathbf{fc}(th)$ is a non-deterministically chosen natural number by the scheduler—for any n, we can use PC-DROP to obtain $\diamond_\kappa(0,\,n)$. Also, by the "yield-fc" rule, whenever the thread th yields and is scheduled again, its fairness counter resets to an *arbitrary* number, breaking the global invariant. Therefore, we require the thread to submit $\diamond_\kappa(1,\,1)$ when it yields, and use it to re-establish the global invariant. The "yield-tax" rule precisely captures this principle.

Lilo provides an abstraction for scheduler fairness, in the form of a predicate € called *scheduler credit*. As we will see in §3.3, we have "yield-sched": we obtain a € whenever a thread returns from a yield, *i.e.,* when it is scheduled. A scheduler credit and the fairness counter roughly has this rule:

$$\forall th\, n.\, (\mathbf{fc}(th) = n) * € \vdash \exists n'.\, (\mathbf{fc}(th) = n') * n' < n$$

Therefore, we can spend a € to decrement the counter of an active thread th. Using this rule and the model of obligation list, we obtain the following rule:

$$((\mathbf{fc}(th) = n * \diamond_\kappa(0,\, n)) \lor P) * € \vdash (\exists n'.\, (\mathbf{fc}(th) = n' * \diamond_\kappa(0,\, n')) * \diamond_\kappa(0,\,1)) \lor P$$

which corresponds to the PROM-PROGRESS rule. A promise $\xrightarrow{\kappa}\!\!\diamond P$ roughly says that $((\mathbf{fc}(th) = n * \diamond_\kappa(0,\, n)) \lor P)$ is in the global invariant, and this rule keeps the global invariant.

However, the user does not need to be aware of this underlying model, and simply use PROM-PROGRESS to enjoy the guarantees of scheduler fairness. The rule requires a scheduler credit € and a promise $\xrightarrow{\kappa}\!\!\diamond P$. Intuitively, a promise $\xrightarrow{\kappa}\!\!\diamond P$ means that there exists some thread th that makes progress towards establishing P. Therefore, PROM-PROGRESS captures the following intuition: whenever thread th is scheduled, it is either making progress for κ (the $\diamond_\kappa(0,\,1)$ case), or th has fulfilled the promise by establishing P (the P case).

Induction principle. In Lilo, when a thread makes some progress for an obligation, other threads can rely on this progress through the induction principle CRED-IND. This rule is governed by the *credit bound* predicate $\blacklozenge_\kappa\lceil \ell,\, n\rceil$, which is persistent knowledge saying that the maximum number of progress credits for κ is $(\ell,\, n)$, as the notation $\lceil \ell,\, n\rceil$ indicates. We obtain $\blacklozenge_\kappa\lceil \ell,\, n\rceil$ when we

create a new obligation κ using CRED-NEW, together with the progress credit $\diamond_\kappa(\ell, n)$. Intuitively, the CRED-IND rule says that the credit bound $\blacklozenge_\kappa\lceil \ell, n\rceil$ *bounds* the number of loop iteration, enabling induction. The rule is a direct consequence of the definition of $\blacklozenge_\kappa\lceil \ell, n\rceil$, and interested readers can refer to our Coq development [2].

The CRED-IND rule roughly says that to prove Q, we can instead prove Q while assuming $(\diamond_\kappa(0, 1) \Rrightarrow Q)$, given $\blacklozenge_\kappa\lceil \ell, n\rceil$. Here, $P \Rrightarrow Q$ is a standard concept in Iris called *view shift* and can be thought of as a logical update from P to Q. The rule needs a persistence modality $\Box$ for soundness, but this is usually not a problem if we set up Q appropriately. Altogether, CRED-IND lets us finish the proof if we can obtain a progress credit $\diamond_\kappa(0, 1)$ and re-establish the proof goal to Q.

As we will demonstrate shortly, we use CRED-IND to prove termination of a *loop*. Specifically, we set up Q to be the simulation weakest precondition for a loop with a loop invariant, and show that we can obtain $\diamond_\kappa(0, 1)$ by iterating the loop once. We remark that CRED-IND provides a significantly simpler interface to reasoning about loops compared to the while-rules of LiLi (Liang and Feng [31, §7.2]) and TaDA Live (D'Osualdo et al. [10, §4.6]). The complexity of while-rules in these logics stems from *scheduler fairness* and *preventing circularity*. In Lilo, scheduler fairness is abstracted away from the user in the form of the scheduler credit and the PROM-PROGRESS rule.

In contrast, circularity in a liveness logic is a problem that the user must address directly: a representative example is a deadlock, where thread 1 waits for thread 2, and thread 2 waits for thread 1, and it is the user's responsibility to prevent such deadlocks. In Lilo, such responsibility surfaces when allocating a new liveness obligation: the user must allocate large enough progress credits for an obligation κ to "pay taxes" for the yields ("yield-tax"). We also need to pay progress credits for κ when carrying out an induction proof for a loop with the CRED-IND rule, which means that if $\blacklozenge_{\kappa'}\lceil \ell, n\rceil$ bounds the loop, we must have greater amount of progress credits for κ than (ℓ, n). This results in a strict order between how the two obligations κ and κ' can depend on each other (κ can depend on κ', but not vice versa), preventing circularity between obligations.

Termination proof of thread 2. Proving the termination of thread 2 requires an induction argument to prove the termination of the while loop, which relies on thread 1's obligation to write 1 to X. This obligation is shared with thread 2 in the form of a promise $\overset{\kappa}{\longrightarrow}\diamond \boxed{X \mapsto 1}^w$. This is obtained from thread 1's initial condition, $Obls_1([(\kappa, \boxed{X \mapsto 1}^w)])$, and PROM-GET.

Concretely, we prove the following thread-local simulations for thread 2 (Sim2):

$$Obls_2([]) * \boxed{\mathrm{MP}_1}^\nu * \overset{\kappa}{\longrightarrow}\diamond \boxed{X \mapsto 1}^w * \blacklozenge_\kappa\lceil 1, 3\rceil \mathrel{-\!\!*} \mathbf{do}\ \{\mathcal{Y}; a \coloneqq X; \mathcal{Y};\}\ \mathbf{while}\ (a = 0); \mathcal{Y}; \mathbf{ret}\ a \precsim^2_\top \mathcal{Y}; \mathbf{ret}\ 1$$

Thread 2 has no obligation, shares the invariant MP_1 with thread 1, and has the credit bound $\blacklozenge_\kappa\lceil 1, 3\rceil$. It also has $\overset{\kappa}{\longrightarrow}\diamond \boxed{X \mapsto 1}^w$, saying that some thread promised to $\boxed{X \mapsto 1}^w$,

We start by applying the induction rule CRED-IND to the do-while loop with $\blacklozenge_\kappa\lceil 1, 3\rceil$ and obtain the following inductive hypothesis (where $loop \triangleq \mathbf{do}\ \{\ \mathcal{Y}; a \coloneqq X; \mathcal{Y};\ \}\ \mathbf{while}\ (a = 0)$):

$$\mathcal{H} \triangleq \diamond_\kappa(0, 1) \Rrightarrow (Obls_2([]) \mathrel{-\!\!*} loop; \mathcal{Y}; \mathbf{ret}\ a \precsim^2_\top \mathcal{Y}; \mathbf{ret}\ 1)$$

Both $\overset{\kappa}{\longrightarrow}\diamond \boxed{X \mapsto 1}^w$ and $\blacklozenge_\kappa\lceil 1, 3\rceil$ are persistent so we can keep them after applying CRED-IND.

We execute the first $\mathcal{Y}$ and open the invariant $\boxed{\mathrm{MP}_1}^\nu$ to get the points-to predicate. The invariant has a disjunction, where the proof for the second case ($shot * \boxed{X \mapsto 1}^w$) is trivial: we have $X \mapsto 1$ so we can terminate the do-while loop and execute the rest of the code to finish the proof.

On the other hand, the first case with $(pend_{1/2} * X \mapsto 0)$ requires an induction argument. In this case, we obtain $a = 0$ from the memory read, close the invariant, and execute $\mathcal{Y}$. At this point, "yield-sched" grants us a scheduler credit €, and since the loop condition $a = 0$ is true, we come back to the original state, $loop; \mathcal{Y}; \mathbf{ret}\ a$.

$$\textsc{inv-alloc}\ \frac{\nu \in \mathcal{N}}{I \Rrightarrow \boxed{I}^{\nu}} \qquad \textsc{inv-pers}\ \frac{}{\mathsf{persistent}(\boxed{I}^{\nu})}$$

$$\textsc{inv-open}\ \frac{\boxed{I}^{\nu} * (I \mathrel{-\!\!*} tgt \ \precsim^{th}_{\mathcal{E}\setminus\{\nu\}} \ src)}{tgt \ \precsim^{th}_{\mathcal{E}} \ src} \qquad \textsc{inv-close}\ \frac{I * \boxed{I}^{\nu} * tgt \ \precsim^{th}_{\mathcal{E}} \ src}{tgt \ \precsim^{th}_{\mathcal{E}\setminus\{\nu\}} \ src}$$

$$\textsc{mem-read}\ \frac{(X \mapsto v) * ((X \mapsto v) \mathrel{-\!\!*} k_t[a \coloneqq v] \ \precsim^{th}_{\mathcal{E}} \ src)}{a \coloneqq X; k_t \ \precsim^{th}_{\mathcal{E}} \ src} \qquad \textsc{mem-write}\ \frac{(X \mapsto v) * ((X \mapsto w) \mathrel{-\!\!*} k_t \ \precsim^{th}_{\mathcal{E}} \ src)}{X \coloneqq w; k_t \ \precsim^{th}_{\mathcal{E}} \ src}$$

$$\textsc{yield-tgt}\ \frac{(\mathit{Obls}_{th}(\Phi) * \mathop{\Large *}_{\kappa\in\mathrm{dom}(\Phi)} \Diamond_{\kappa}(1,\,1)) * ((\mathit{Obls}_{th}(\Phi) * \text{€}) \mathrel{-\!\!*} k_t \ \precsim^{th}_{\top} \ \mathcal{Y}; k_s)}{\mathcal{Y}; k_t \ \precsim^{th}_{\top} \ \mathcal{Y}; k_s} \qquad \textsc{sim-term}\ \frac{\mathit{Obls}_{th}([]) \quad r_t = r_s}{\mathbf{ret}\ r_t \ \precsim^{th}_{\top} \ \mathbf{ret}\ r_s}$$

Fig. 3. Selected and simplified simulation rules and invariant rules.

Now, we use € with $\xrightarrow{\kappa}\Diamond\boxed{X \mapsto 1}^{w}$ to obtain $\Diamond_{\kappa}(0,\,1) \vee \boxed{X \mapsto 1}^{w}$ through PROM-PROGRESS. Again, the proof becomes trivial with the second case $\boxed{X \mapsto 1}^{w}$, because we know $X \mapsto 1$ which is enough to terminate the loop. For the other case with $\Diamond_{\kappa}(0,\,1)$, we use the inductive hypothesis $\mathcal{H}$ together with $\Diamond_{\kappa}(0,\,1)$ to finish the proof by induction.

3.3 Thread-Local Relational Reasoning with Simulation Weakest Precondition

Lilo enables thread-local reasoning through simulation weakest precondition, which supports Iris-style invariants [22, 24] that is significantly more flexible compared to the invariants in Fairness Logic. Note that Lilo's invariant system is implemented with stratified propositions, inspired by Nola [32]. We discuss stratified propositions in §5 and abstract them away for now, presenting simplified proof rules of Lilo in this subsection (Fig. 3).

Invariant rules. Invariants enforce user-defined protocols on the shared state through rely-guarantee reasoning—in the simulation proof, we can *rely* on invariants associated with $\mathcal{E}$ when the thread receives control (*e.g.,* after $\mathcal{Y}$) and must *guarantee* invariants associated with $\mathcal{E}$ when it passes control (*e.g.,* before $\mathcal{Y}$). We can add a proposition I to invariants with a name ν using the INV-ALLOC rule, and any invariant is persistent, *i.e.,* , stays true forever, once it is allocated.

Once I is added to invariants, we can rely on I using the INV-OPEN rule: the rule says that we can *open* the mask ($\mathcal{E}$ to $\mathcal{E} \setminus \{\nu\}$) to obtain I. On the other hand, we must establish I and use the INV-CLOSE rule to *close* the mask ($\mathcal{E} \setminus \{\nu\}$ to $\mathcal{E}$), which guarantees the invariant I. We use the $\top$ mask to represent that every invariant used in the proof is established and that the mask is completely closed. Finally, we note that unlike Iris invariant rules, our invariant rules don't have a later modality because we do not use step-indexing.

Simulation rules. Most of the rules for proving the simulation weakest precondition are standard [13, 28]. For example, to execute the memory read operation from X, we can use the MEM-READ rule with a points-to predicate $X \mapsto v$, which returns the points-to predicate and brings us to the next instruction while replacing the local variable a with the value v. On the other hand, we can execute the memory write operation to X using MEM-WRITE, which requires a points-to predicate $X \mapsto v$, executes the write operation, and returns the points-to predicate with the written value.

The central simulation rule of Lilo is the YIELD-TGT rule, which is a simulation proof rule to execute a yield $\mathcal{Y}$ in the target. YIELD-TGT serves two main purposes.

The first purpose is to ensure that the rely-guarantee nature of invariants is enforced. When executing YIELD-TGT, it requires a $\top$ mask, which forces one to close (guarantee) all invariants before yielding control back to the scheduler. Conversely, we can *open* (rely on) any invariant when the thread receives the control back because the rule ensures the $\top$ mask ($\precsim^{th}_{\top}$ in the premise).

The second purpose is to enable reasoning about liveness and scheduler fairness, *i.e.*, YIELD-TGT validates the "yield-tax" and "yield-sched" rules introduced earlier. Specifically, when executing YIELD-TGT, we need to submit $\diamond_\kappa(1, 1)$ for each obligation in the obligation list $\mathit{Obls}_{th}(\Phi)$ of the executing thread, where we use $\mathrm{dom}(\Phi)$ to denote the list of obligation ids in Φ. This corresponds to the "yield-tax" rule. In addition, the thread receives a scheduler credit € when it resumes its execution after it returns from the yield. This corresponds to the "yield-sched" rule.

To finish off a simulation, the return rule SIM-TERM says that simulation between two return statements holds only when the return values are the same and the obligation list is *empty*. Requiring an empty obligation list prevents threads from terminating without fulfilling their obligations, which would break the rely-guarantee reasoning of the obligation list and promise.

Adequacy. Adequacy of Lilo is a consequence of the adequacy theorem of Fairness Logic (Lee et al. [28, Theorem 7.2]) and the algebra of the underlying PCMs of the liveness assertions. Specifically, the simulation weakest precondition of Lilo is developed on top of Fairness Logic by developing PCMs and global invariants for Lilo invariants and obligation lists. Fairness Logic itself is developed on top of the thread-local simulation relation of FOS, which implies termination-preserving fair refinement. Altogether, Lilo proves fair refinements, denoted $\sqsubseteq$, as the following (simplified) theorem states:

THEOREM 3.1 (ADEQUACY). *For source and target threads s_i and t_i with thread ids $i \in K = \{1, \ldots, k\}$, if the initial physical and ghost states satisfy the initial invariant, we have*

$$\ast_{i\in K} \mathit{Obls}_i([]) \Rrightarrow\!\ast_\top (\ast_{i\in K}\, t_i \precsim^i_\top s_i) \Rightarrow t_1 \parallel \ldots \parallel t_k \sqsubseteq s_1 \parallel \ldots \parallel s_k$$

where $a_1 \parallel \ldots \parallel a_k$ denotes the behavior of the composition of threads a_i.

We note that the adequacy theorem allows us to perform thread-local simulations: to prove the refinement $\sqsubseteq$ between two programs, we can prove $\precsim^i_\top$ of each thread separately. Also, the theorem allows us to distribute preconditions to each thread through the view shift $\Rrightarrow\!\ast_\top$.

To demonstrate this, we briefly discuss how to prove $\mathrm{MP} \sqsubseteq \mathrm{MP_S}$ using the two results Sim1 and Sim2. We first apply Theorem 3.1 to $\mathrm{MP} \sqsubseteq \mathrm{MP_S}$ and get $\ast_{i\in\{1,2\}} \mathit{Obls}_i([]) \Rrightarrow\!\ast_\top (\ast_{i\in\{1,2\}}\, t_i \precsim^i_\top s_i)$ where t_i and s_i are the threads of MP and $\mathrm{MP_S}$. Then, from the initial state, we derive preconditions of Sim1 and Sim2, which is possible thanks to the view shift. In particular, we use INV-ALLOC to obtain the invariant $\boxed{\mathrm{MP_I}}^\nu$ from the initial state, and we use rules of liveness obligations such as CRED-NEW, OBLS-ADD, and PROM-GET to obtain liveness-related predicates such as $\mathit{Obls}_1([(\kappa, \boxed{X \mapsto 1}^w)])$. Finally, we apply the simulation proofs Sim1 and Sim2 to finish the proof.

4 Modular Specification for Liveness

Lilo's liveness reasoning based on progress credits enables modular specifications for library functions with both safety and liveness guarantees. As we discuss in §4.1, a modular specification for liveness must concern two classes of effects that influence progress of threads, each called *blocking* and *delay* (also called *impedance*) [10, 30]. In §4.2, we show how Lilo enables reasoning about blocking and delay through a novel proof technique called *obligation links*. Finally, we discuss what is leaked by Lilo's specifications and its consequences in §4.3.

4.1 Motivating Example: When Does a Spinlock Terminate?

Spinlock guarantees exclusive access to shared resources such as shared memory locations. It achieves exclusiveness with a **CAS** operation as follows:

$$\texttt{def lock}(x) = \texttt{do}\ \{\ \mathcal{Y}; b \coloneqq \texttt{CAS}(x, 0, 1);\ \}\ \texttt{while}\ (b = 0); \qquad \texttt{def unlock}(x) = \mathcal{Y}; x \coloneqq 0; \quad \text{(Spinlock)}$$

When a thread acquires and owns the lock, the spinlock ensures exclusive access to the owner by forcing other threads to wait in a loop until the lock is unlocked. This protocol is implemented

using a $\mathbf{CAS}(x, old, new)$, which compares the value at x with old and if they are the same, writes new to x and returns 1. If the value at x is not the same as old, $\mathbf{CAS}(x, old, new)$ simply returns 0 and leaves x as is. Consequently, a thread can successfully write 1 to x and exit the do-while loop of $\texttt{lock}(x)$ *only when* the value at x is 0, which indicates that the lock is not acquired by any other thread. To release a lock, a thread can invoke $\texttt{unlock}(x)$ that simply writes 0 to x.

A $\texttt{lock}(x)$ invocation may not terminate due to its do-while loop. More concretely, there are two scenarios in which an invocation may not terminate.

Scenario 1: missing unlock. If a thread acquires the lock x but does not release it, another thread's invocation of $\texttt{lock}(x)$ does not terminate. In this case, x is fixed to 1 and $\mathbf{CAS}(x, 0, 1)$ always returns 0, and any waiting thread cannot exit the loop. We say that waiting threads are *blocked* by the lock.

Scenario 2: infinite lock. If a thread acquires the lock x infinitely many times before another thread acquires it, another thread's $\texttt{lock}(x)$ may not terminate. This is because the $\mathbf{CAS}(x, 0, 1)$ operation does not guarantee liveness: when two threads are competing for a $\mathbf{CAS}(x, 0, 1)$, it is possible that only the first thread successfully swaps the value and gets the return value 1, starving the other thread. Scheduler fairness does not help here because the second thread may get its turns only when the first thread is holding the lock. We say that waiting threads are *delayed*.

Termination of a spinlock. Once the above two scenarios are prevented, we can prove the termination of $\texttt{lock}(x)$ by the following high-level arguments. "Unblock": Whenever a thread acquires the lock, it will *eventually* unlock it, *i.e.,* unblock waiting threads. "Finite delay": Also, since the lock is acquired only a *finite* number of times, waiting threads are delayed only finite times and eventually becomes the *only* thread waiting for the lock. "Terminate": Then, since the lock is eventually unlocked by "Unblock", the thread can acquire the lock.

4.2 Proving a Specification of a Spinlock

Basic specification. We present Lilo's Hoare triple for Spinlock that guarantees termination:

$$\begin{aligned}\mathrm{isSL}(\kappa_S, x, L, N) \vdash \{&\mathit{Obls}_{th}([]) * \Diamond_{\kappa_S}(\ell, n+1)\} \quad \texttt{lock}(x) \\ &\{\exists \kappa_U.\, \mathit{Obls}_{th}([(\kappa_U, \ulcorner \mathit{shot} \urcorner^{\gamma})]) * \Diamond_{\kappa_U}(\ell, n) * \mathit{locked}(\kappa_U)\}_{\top}\end{aligned} \tag{HT-SL}$$

Here, $\mathit{locked}(\kappa_U)$ is a token that represents the exclusive access granted to the owner of the lock (we omit ghost locations for brevity). This is a basic specification because it assumes *empty* client obligation $\mathit{Obls}_{th}([])$. We later show how we can extend it to handle general client obligations.

In Lilo, the Hoare triple is simply a format for writing the simulation weakest precondition:[3]

$$\{P\}\ f(x)\ \{v.\, Q(v)\}_{\mathcal{E}} \triangleq \forall k_t\, k_s.\, (P * (\forall v.\, Q(v) \mathrel{-\!\!*} k_t(v) \lesssim^{th}_{\mathcal{E}} \mathcal{Y}; k_s)) \mathrel{-\!\!*} f(x); k_t \lesssim^{th}_{\mathcal{E}} \mathcal{Y}; k_s$$

The definition says that for *arbitrary target and source continuations* k_t and k_s, the verifier obtains $f(x); k_t \lesssim^{th}_{\mathcal{E}} \mathcal{Y}; k_s$ if the verifier proves the *precondition* P and the simulation $(\forall v.\, Q(v) \mathrel{-\!\!*} k_t(v) \lesssim^{th}_{\mathcal{E}} \mathcal{Y}; k_s)$ for the continuations given the *postcondition* Q. This definition preserves termination as it builds on a termination-preserving simulation. Also, the Hoare triple requires a $\mathcal{Y}$ in the source side because target $\mathcal{Y}$s can only be executed when the source side also has a $\mathcal{Y}$ (YIELD-TGT).

The specification HT-SL mentions two obligations, κ_U and κ_S, each appearing in the postcondition and the precondition. The κ_U obligation captures the "Unblock" argument that ensures that every lock is eventually unlocked. HT-SL enforces this by *adding a new obligation* κ_U to the obligation list in the postcondition, $\mathit{Obls}_{th}([(\kappa_U, \ulcorner \mathit{shot} \urcorner^{\gamma})])$, where the promised predicate $\ulcorner \mathit{shot} \urcorner^{\gamma}$ can be fulfilled by calling the unlock function. Also, the thread that acquires the lock gets $\Diamond_{\kappa_U}(\ell, n)$ that can be

[3] Our approach follows Iris's Hoare triple, which includes a persistence modality. For our examples, attaching a persistence modality makes no difference, and we choose to omit it for brevity.

LINK-PERS
persistent($\kappa_1 \multimap\!\!\diamond\, \kappa_2$)

LINK-NEW
$$\frac{\blacklozenge_{\kappa_1}\lceil \ell,\, n\rceil \;*\; \diamond_{\kappa_2}(\ell,\, n)}{\Rightarrow\!\!\!\!\cdot\; \kappa_1 \multimap\!\!\diamond\, \kappa_2}$$

LINK-AMP
$$\frac{\diamond_{\kappa_1}(\ell,\, n) \;*\; \kappa_1 \multimap\!\!\diamond\, \kappa_2}{\Rightarrow\!\!\!\!\cdot\; \diamond_{\kappa_2}(\ell,\, n)}$$

LINK-TRANS
$$\frac{\kappa_1 \multimap\!\!\diamond\, \kappa_2 \;*\; \kappa_2 \multimap\!\!\diamond\, \kappa_3}{\kappa_1 \multimap\!\!\diamond\, \kappa_3}$$

Fig. 4. Selected and simplified rules of obligation link.

spent to execute the yields before it unlocks the lock, where the parameters ℓ and n are decided by the client when initializing the lock. For example, if the thread needs to execute three yields, we must instantiate at least $\ell = 1$ and $n = 3$, and we use PC-SPLIT to get three $\diamond_{\kappa_U}(1,\, 1)$s to execute the yields. If we instead instantiate $\ell = 1$ and $n = 2$, we cannot execute the third yield.

The κ_S obligation captures the "Finite delay" argument that ensures that the lock is acquired only finite times. Note that unlike κ_U, this obligation is not tied to a specific thread, because "Finite delay" is a logical constraint that is imposed by the *programming pattern*. HT-SL enforces constraint by requiring $\diamond_{\kappa_S}(\ell,\, n+1)$ in the precondition. Because there exists only *finite* number of progress credits for κ_S, we can satisfy the precondition only finite number of times. This ensures that the lock function is called only finite times, guaranteeing finite delay. Here, the amount $(\ell,\, n+1)$ is related to the proof structure of the specification, which we explain shortly.

Obligation link. Formally, we can prove the specification, *i.e.,* establish "Terminate", with a *nested induction* on the do-while loop, where the outer induction corresponds to "Finite delay" and the inner induction corresponds to "Unblock". More concretely, because the lock is acquired only a finite number of times, we do the first induction on the number of lock acquisitions. If the lock is already unlocked, we can just acquire it. If the lock is acquired by some other thread, we do the second induction on the condition that the lock is eventually unlocked.

In Lilo, such a nested induction proof structure is captured by an *obligation link* (Fig. 4). An obligation link $\kappa_1 \multimap\!\!\diamond\, \kappa_2$ is obtained using LINK-NEW, which says that if the *maximum possible number of progress credits* for κ_1 is $(\ell,\, n)$, as denoted by $\blacklozenge_{\kappa_1}\lceil \ell,\, n\rceil$, and if we provide equal amount of progress credits for κ_2, $\diamond_{\kappa_2}(\ell,\, n)$, we obtain persistent knowledge $\kappa_1 \multimap\!\!\diamond\, \kappa_2$. This knowledge says that whenever we have $\diamond_{\kappa_1}(\ell_0,\, n_0)$, we can convert it into $\diamond_{\kappa_2}(\ell_0,\, n_0)$ using LINK-AMP. This is sound because we already submitted enough amount of credits for κ_2 through LINK-NEW. Obligation links are transitive, LINK-TRANS, as expected from its meaning.

The rule LINK-AMP captures the nested induction reasoning: κ_1 corresponds to the inner induction and κ_2 to the outer induction. For the spinlock example, the nested induction is captured by $\kappa_U \multimap\!\!\diamond\, \kappa_S$. Whenever a thread with the lock makes progress to unlocking the lock (κ_U), this also means that the overall program is making progress towards reaching the last turn of the finite locking (κ_S). To carry out this reasoning, we must allocate large enough progress credits for κ_S that can be used for every new lock acquisition. As we see in the specification HT-SL, a new obligation κ_U is created whenever a lock is acquired, and therefore a new $\kappa_U \multimap\!\!\diamond\, \kappa_S$ for this κ_U must also be created. We will soon see that we allocate $\blacklozenge_{\kappa_U}\lceil \ell,\, n+1\rceil$ for the new κ_U, so to create $\kappa_U \multimap\!\!\diamond\, \kappa_S$ with LINK-NEW, we need $\diamond_{\kappa_S}(\ell,\, n+1)$, which is why the specification requires it in the precondition.

Moreover, obligation links enable induction on an obligation that *no thread is obligated to fulfill*. The basic principle of induction in Lilo involves selecting an obligation κ and combining the rules CRED-IND and PROM-PROGRESS. In particular, we need a promise for κ to use the PROM-PROGRESS rule, and a promise for κ indicates that some thread is obligated to fulfill that promise. However, the explicit requirement of a promise is too restrictive in many cases: for example, the spinlock example's finite locking obligation κ_S is not assigned to any thread, and we cannot obtain a promise for κ_S. This prevents us from directly using PROM-PROGRESS with κ_S when we try to do induction with κ_S, but because we have an obligation link $\kappa_U \multimap\!\!\diamond\, \kappa_S$, we can instead use PROM-PROGRESS *with* κ_U and LINK-AMP with $\kappa_U \multimap\!\!\diamond\, \kappa_S$. This combination of rules can give us a progress credit for κ_S, which we can use with CRED-IND to do an induction proof.

Invariant. We use the following invariant for HT-SL (we omit details related to ghost locations):

$$\mathrm{isSL}(\kappa_S, x, L, N) \triangleq \boxed{\mathrm{SL_I}(\kappa_S, x)}^{\nu} * \blacklozenge_{\kappa_S}\lceil L, N\rceil \qquad \mathit{locked}(\kappa_U) \triangleq \boxed{\circ(\underline{\kappa_U})} \tag{SL$_\mathrm{I}$}$$

$$\mathrm{SL_I}(\kappa_S, x) \triangleq \exists \kappa_U.\, \bullet(\kappa_U) * (x \mapsto 0 * \mathit{locked}(\kappa_U)) \vee (x \mapsto 1 * \boxed{\mathit{pend}}^{\gamma} * \overset{\kappa_U}{\Diamond}\boxed{\mathit{shot}}^{\gamma} * \kappa_U \multimap \kappa_S)$$

The invariant $\mathrm{SL_I}(\kappa_S, x)$ describes a two-state transition system, where the state with $x \mapsto 0$ represents that the lock is *unlocked* and the other state with $x \mapsto 1$ represents that the lock is *locked* by some owner thread. In the invariant, κ_U ensures finite blocking where the promise $\overset{\kappa_U}{\Diamond}\boxed{\mathit{shot}}^{\gamma}$ is fulfilled by unlocking the lock. Also, κ_S ensures finite delay, where $\blacklozenge_{\kappa_S}\lceil L, N\rceil$ indicates the maximum number of delays possible. Moreover, we use the usual authoritative assertions [22] $\bullet(\kappa_U)$ and $\circ(\kappa_U)$ to resolve the existential variable κ_U.

The persistent predicate $\mathrm{isSL}(\kappa_S, x, L, N)$ packs the invariant with $\blacklozenge_{\kappa_S}\lceil L, N\rceil$. We use $\blacklozenge_{\kappa_S}\lceil L, N\rceil$ to carry out an induction proof with the CRED-IND induction rule. However, as we discussed, termination of spinlock involves a nested proof structure, which is captured by the obligation link $\kappa_U \multimap \kappa_S$. Therefore, even if κ_U makes progress instead of κ_S, *i.e.,* we obtain $\Diamond_{\kappa_U}(0, 1)$, we can convert it into progress of κ_S and carry out induction. This significantly reduces proof effort, because identifying nested structures and constructing adequate induction hypotheses presents the main challenge in liveness proofs, especially in FOS (Lee et al. [28, §9]).

Proof. Proving that the lock function of Spinlock satisfies HT-SL boils down to proving the termination of the while loop, *i.e.,* the "Terminate" argument. We first set up an induction hypothesis using CRED-IND with $\blacklozenge_{\kappa_S}\lceil L, N\rceil$. Thanks to the obligation link, this single induction is enough to carry out the nested argument. Then, we open the invariant $\mathrm{SL_I}$ and get the two cases.

If we are in the $x \mapsto 1$ case, we iterate once where YIELD-TGT gives us a scheduler credit €. Then, PROM-PROGRESS with € and $\overset{\kappa_U}{\Diamond}\boxed{\mathit{shot}}^{\gamma}$ gives us $\Diamond_{\kappa_U}(0, 1) \vee \boxed{\mathit{shot}}^{\gamma}$, where the case with $\boxed{\mathit{shot}}^{\gamma}$ is eliminated by contradiction with $\boxed{\mathit{pend}}^{\gamma}$. Therefore, we get $\Diamond_{\kappa_U}(0, 1)$ and use this with $\kappa_U \multimap \kappa_S$ to obtain $\Diamond_{\kappa_S}(0, 1)$. We use this with the inductive hypothesis to finish the proof by induction.

If we are in the other case with $x \mapsto 0$, we execute the $\mathbf{CAS}(x, 0, 1)$ and get $b = 1$ with $x \mapsto 1$, which means the loop terminates. To close the invariant, we must allocate a new oneshot predicate $\boxed{\mathit{pend}}^{\gamma}$. In addition, we also allocate a new "unblock" obligation κ_U with CRED-NEW and obtain $\blacklozenge_{\kappa_U}\lceil \ell, n+1\rceil$ and $\Diamond_{\kappa_U}(\ell, n+1)$. We use PC-SPLIT and PC-DROP to get $\Diamond_{\kappa_U}(\ell, n)$ and $\Diamond_{\kappa_U}(1, 1)$, where the former is returned as the postcondition. The latter, $\Diamond_{\kappa_U}(1, 1)$, is consumed by OBLS-ADD to add κ_U with a promise to fulfill $\boxed{\mathit{shot}}^{\gamma}$ in the obligation list. Then, we use PROM-GET to obtain a promise $\overset{\kappa_U}{\Diamond}\boxed{\mathit{shot}}^{\gamma}$. Moreover, we use LINK-NEW with $\blacklozenge_{\kappa_U}\lceil \ell, n+1\rceil$, which we just allocated, and $\Diamond_{\kappa_S}(\ell, n+1)$, which is given as precondition, to get $\kappa_U \multimap \kappa_S$. Finally, we update $\bullet(\kappa_U)$ and $\circ(\kappa_U)$, close the invariant, and satisfy the postcondition to finish the proof.

The unlock obligation κ_U can be fulfilled by calling $\mathtt{unlock}(x)$, which has the following spec:

$$\mathrm{isSL}(\kappa_S, x, L, N) \vdash \{\mathit{Obls}_{th}([(\kappa_U, \boxed{\mathit{shot}}^{\gamma})]) * \Diamond_{\kappa_U}(1, 1) * \mathit{locked}(\kappa_U)\}\ \mathtt{unlock}(x)\ \{\mathit{Obls}_{th}(\varnothing)\}_{\top}$$

The precondition requires an obligation list with the unlock obligation κ_U and the $\mathit{locked}(\kappa_U)$ token, and also a credit $\Diamond_{\kappa_U}(1, 1)$ which we need to execute a $\lightning$ in the $\mathtt{unlock}(x)$. Then the spec fulfills the unlock obligation and returns an empty obligation list $\mathit{Obls}_{th}([])$.

Client obligations. We can generalize HT-SL to deal with client obligations, where the thread has non-empty obligations $\mathit{Obls}_{th}(\Phi)$. In this case, the precondition requires $\Diamond_{\kappa}(L+1, N+1)$ for each obligation in Φ, so that we can spend $\divideontimes_{\kappa \in \mathrm{dom}(\Phi)} \Diamond_{\kappa}(1, 1)$ to execute $\lightning$s while we are waiting for the lock in the loop. The number $(L+1, N+1)$ roughly captures the *maximum number of yields* a thread might execute to acquire the spinlock. As we do induction with $\blacklozenge_{\kappa_S}\lceil L, N\rceil$, the maximum number of iteration roughly corresponds to (L, N). There are fixed number of yields to execute for

each iteration, and around $(L+1,\ N+1)$ is large enough compared to the total number of yields. Note that we did not optimize the conditions for a tighter bound.

4.3 Limitation of Lilo Specifications

The previous discussion reveals that a spinlock specification with client obligations *leaks the number of yields* of the function implementation, in the form of requiring progress credits for client obligations. In particular, the layer leaks in Lilo proofs, and the user must concern about layers when allocating obligations or initializing specifications. This is usually straightforward because the user can pick large enough layers for obligations, and layers in specifications are parametrized. Nonetheless, the leakage of layer information poses two issues regarding Lilo's modularity.

First issue is that the number of yields can change if the implementation is updated, and we might need to update the specification accordingly. Thankfully, Lilo's abstractions provide reasonable robustness against such updates. In particular, we can set up a large enough layer to tolerate updates that add static number of yields. For spinlock, a specification that requires $\diamond_\kappa(L+2,\ N+1)$ for client obligations can tolerate updates that add a fixed number of yields inside the while loop.

Second issue is that Lilo's Hoare triples cannot capture functions involving *unbounded dynamic nesting* of blocking functions. We probe this issue using a ticket lock, which is a *starvation free* lock, and unlike a spinlock, it does not require "Finite delay" for termination. This significantly simplifies our discussion, so we use ticket locks instead of spinlocks here.

Nesting ticket locks. We present a specification of a ticket lock (see §7.1 for the code):

$$\mathrm{isTL}(x,\ \ell)\ \vdash \{\mathit{Obls}_{th}(\Phi)\ *\ \mathop{\Large *}_{\kappa\in\mathrm{dom}(\Phi)} \diamond_\kappa(\ell+4,\ 1)\}\quad \mathtt{lock_{tk}}(x) \qquad \text{(HT-TL)}$$
$$\{\exists \kappa_U.\, \mathit{Obls}_{th}((\kappa_U, \overset{\lceil}{\underset{\lfloor}{\mathit{shot}}}{}^{\gamma})::\Phi)\ *\ \diamond_{\kappa_U}(\ell,\ 1)\ *\ \mathit{locked}(\kappa_U)\}_\top$$

This is similar to the generalized specification of a spinlock, but without dealing with the finite delay. We assume a ticket lock invariant $\mathrm{isTL}(x,\ \ell)$, and the precondition requires $(\ell+4,\ 1)$ progress credits for obligations in Φ. Then, the postcondition adds a new obligation κ_U to unlock the lock to Φ and gives $\diamond_{\kappa_U}(\ell,\ 1)$ and a $\mathit{locked}(\kappa_U)$ token. We note that $(\ell+4,\ 1)$ is a loose bound that is more than enough to cover the number of yields a thread needs to execute to acquire the ticket lock. Moreover, unlike a spinlock's specification HT-SL, HT-TL does not require progress credits in its precondition to ensure finite delay, because ticket lock is starvation free.

Then, consider the following function called `incrBoth` by D'Osualdo et al. [10, §5.4]:

$$\begin{aligned}\texttt{def incrBoth}(l_a,\ l_b,\ x,\ y)\ =& \mathcal{Y};\ \mathtt{lock_{tk}}(x);\ \mathcal{Y};\ \mathtt{lock_{tk}}(y);\ \mathcal{Y};\ a \coloneqq l_a;\ \mathcal{Y};\ l_a \coloneqq a+1; \qquad \text{(INCR-B)}\\ & \mathcal{Y};\ b \coloneqq l_b;\ \mathcal{Y};\ l_b \coloneqq b+1;\ \mathcal{Y};\ \mathtt{unlock_{tk}}(y);\ \mathcal{Y};\ \mathtt{unlock_{tk}}(x);\end{aligned}$$

This uses a *nested ticket lock* to increment values of two shared memory locations l_a and l_b. Then, consider developing a specification of INCR-B using that of ticket locks, HT-TL. According to HT-TL, the outer lock introduces an unlock obligation κ_{Ux}, which can only be fulfilled *after* acquiring and releasing the inner lock. For INCR-B, the inner lock needs $\diamond_{\kappa_{Uy}}(2,\ 1)$ to fulfill its unlock obligation κ_{Uy}, so we have $\mathrm{isTL}(y,\ 2)$. Therefore, for κ_{Ux}, we need $\diamond_{\kappa_{Ux}}(2+4,\ 1)$ to execute yields inside $\mathtt{lock_{tk}}(y)$, and $\diamond_{\kappa_{Ux}}(2,\ 1)$ to execute innermost and auxiliary yields.

Altogether, we have the following specification for INCR-B:

$$\mathrm{isTL}(x,\ 7) * \mathrm{isTL}(y,\ 2)\ \vdash \{\mathit{Obls}_{th}(\Phi)\ *\ \mathop{\Large *}_{\kappa\in\mathrm{dom}(\Phi)} \diamond_\kappa(12,\ 1)\ *\ l_a \mapsto a\ *\ l_b \mapsto b\}\quad \mathtt{incrBoth}(l_a,\ l_b,\ x,\ y)$$
$$\{\mathit{Obls}_{th}(\Phi)\ *\ l_a \mapsto a+1\ *\ l_b \mapsto b+1\}_\top$$

The precondition leaks the total number of yields inside the INCR-B function by requiring $\mathop{\Large *}_{\kappa\in\mathrm{dom}(\Phi)} \diamond_\kappa(12,\ 1)$. Also, $\mathrm{isTL}(y,\ 2)$ shows that the function executes $(2,\ 1)$ yields after acquiring the inner lock, and $\mathrm{isTL}(x,\ 7)$ shows that the function executes $(7,\ 1)$ yields after acquiring the

outer lock. Note that if we do not want to expose these implementation details, we can hide $\mathrm{isTL}(x,\ 7) * \mathrm{isTL}(y,\ 2)$ by defining $\mathrm{isIncrBoth}(x,\ y)$. However, we cannot hide $\divideontimes_{\kappa \in \mathrm{dom}(\Phi)} \diamond_\kappa (12,\ 1)$ because client obligations must be aware of the number of yields they must execute.

As our discussion reveals, the *layer* of required progress credits for client obligations *blows up* when we nest *blocking* functions. This is inevitable because nesting blocking functions introduces a structure similar to nested loops, where the number of yields is multiplied for each nest. This blow up can be addressed as long as only a fixed number of blocking functions are nested: INCR-B example shows that we can compute a large enough layer that can cover the total number of yields.

However, this abstraction fails when functions involve *unbounded dynamic nesting* of blocking functions, because we cannot determine the upper bound of the number of yields we must execute. For example, a lock-coupling set [10, §5.5] nests N locks, where N corresponds to the length of the linked list used to implement the set. Because N is an implementation detail, we want to hide it from the specification. Then, the precondition of a lock-coupling set requires $(M, 1)$ progress credits for client obligations, where M is an arbitrary natural number depending on the dynamic internal information of the implementation. Lock-coupling set is an especially complicated example, and TaDA Live also required nontrivial extensions to prove a specification for it. We believe generalizing the layer of progress credits to ordinal numbers can solve this limitation, which we leave as a future work together with the verification of lock-coupling set.

5 Stratified Propositions for Higher-Order Reasoning

Lilo relies on *higher-order reasoning techniques* such as *higher-order ghost states* [21] for various constructions, such as invariants and obligation lists. To support higher-order reasoning, we develop *stratified propositions*, inspired by Nola [32].

Issue with step-indexing. Although step-indexing is used heavily in higher-order separation logic for *safety* [21], it is known to be ill-suited for liveness reasoning [14, 40, 45] because step-indexing "restricts reasoning to finite prefixes of program execution," as discussed by Timany et al. [45, §1, §2.2]. Existing step-indexing based approaches sidestep this limitation in various ways. Trillium [45] utilizes the fact that liveness properties can be approximated by safety properties by imposing restrictions. For example, termination is a liveness property, but termination with a fixed upper bound on the number of steps is a safety property. Simuliris [14] supports liveness but does not use step-indexing, and consequently, only supports static invariants. Spies et al. develop *transfinite* step-indexing that supports liveness, but it is not yet applied to concurrency. The goal of Lilo is proving true liveness properties, such as fair refinement of non-terminating programs, and we do not use step-indexing to avoid the said issue. Developing a step-indexing based concurrent separation logic for liveness is not in the scope of our work.

5.1 Definition of Stratified Propositions

Stratified propositions sProp_i are *stratified syntaxes* for the language of logical assertions in Lilo, where the natural number i is the stratification index (Fig. 5). The term "stratified" comes from the fact that we have a *set* of syntaxes, *i.e.,* sProp_0, sProp_1, sProp_2, *etc.*, to express logical assertions in Lilo. The definition of stratified propositions sProp_i consists of types τ of terms in sProp_i, the interpretation $\mathcal{I}(\tau,\ i)$ of sProp_i types to the meta-level type ***Type*** (*i.e.,* "Type" type of Coq), the stratified syntaxes of logical assertions sProp_i, typically written as P_i or Q_i, and the interpretation $[\![\cdot]\!]_i$ of sProp_i to *iProp*, the semantic domain of stratified propositions.

Types in sProp_i and their interpretation. The types τ in sProp_i are part of the syntax of Lilo assertions for representing the actual semantic interpretation of those types. For example, the type of the natural numbers $\mathbb{N}$ in sProp_i is a syntax and its interpretation $\mathcal{I}(\mathbb{N},\ i)$ (ignore i for now)

$$\tau \triangleq \phi \mid \mathbb{N} \mid \mathbb{B} \mid \mathbb{Z} \mid \ldots \mid \tau \to \tau \mid \tau \times \tau \mid \tau + \tau \mid \ldots$$
$$\mathcal{I}(\tau, i) \triangleq \textbf{match}\ \tau\ \textbf{with} \mid \phi \Rrightarrow sProp_i \mid \mathbb{N} \Rrightarrow \mathbb{N} \mid \ldots$$
$$\mid \tau \to \tau \Rrightarrow \mathcal{I}(\tau, P_i) \to \mathcal{I}(\tau, P_i) \mid \ldots$$
$$P_i, Q_i \triangleq \mathsf{True} \mid \mathsf{False} \mid P_i \wedge Q_i \mid P_i \vee Q_i \mid P_i \Rightarrow Q_i$$
$$\mid P_i * Q_i \mid P_i \mathrel{-\!\!*} Q_i \mid \Box P_i \mid \Rightarrow\!\!\!\!\!\!\cdot\ P_i \mid \boxed{\ulcorner \alpha \urcorner}^{\gamma}$$
$$\mid \forall_\tau (x : \mathcal{I}(\tau, i-1)).\, P_i \mid \exists_\tau (x : \mathcal{I}(\tau, i-1)).\, P_i$$
$$\mid \uparrow P_{i-1} \mid \boxed{P_i}^{\nu} \mid Obls_{th}(\Phi) \mid \xrightarrow{\kappa}\!\!\!\!\!\!\diamond\ P_i \mid \ldots$$

$$\Phi : \kappa \xrightarrow{\text{fin}} P_i \qquad \mathcal{I}(\tau, i) : \textbf{\textit{Type}}$$
$$[\![\cdot]\!]_i : sProp_i \to iProp \qquad sProp_0 = \varnothing$$
$$[\![P_i]\!]_i \triangleq \textbf{match}\ P_i\ \textbf{with} \mid \ldots \mid \boxed{\ulcorner \alpha \urcorner}^{\gamma} \Rrightarrow \boxed{\ulcorner \alpha \urcorner}^{\gamma}$$
$$\mid \forall_\tau (x : \mathcal{I}(\tau, i-1)).\, P_i \Rrightarrow \forall x.\, [\![P_i\, x]\!]_i$$
$$\mid \exists_\tau (x : \mathcal{I}(\tau, i-1)).\, P_i \Rrightarrow \exists x.\, [\![P_i\, x]\!]_i$$
$$\mid \uparrow P_{i-1} \Rrightarrow [\![P_{i-1}]\!]_{i-1} \mid \boxed{P_i}^{\nu} \Rrightarrow \boxed{P_i}^{\nu}$$
$$\mid Obls_{th}(\Phi) \Rrightarrow Obls_{th}(\Phi) \mid \ldots$$

Fig. 5. Simplified definition of types, type interpretation, syntax, and interpretation of $sProp_i$.

is the actual Coq type $\mathbb{N}$ of the natural numbers. The types in $sProp_i$ can include syntax for any semantic type unless the semantic type *depends* on an *arbitrary* $sProp_i$ type; for instance, types τ cannot include a syntax $sProp_3$ whose interpretation is defined as $\mathcal{I}(sProp_3, i) = sProp_3$.

This restriction is essential for allowing *quantifications* over a type τ in $sProp_i$, which depends on the interpretation of types. For instance, universal quantification in $sProp_i$ has the following form: $\forall_\tau (x : \mathcal{I}(\tau, i-1)).\, P_i$. The $\forall$ quantifies over the *interpretation* of the type τ and the assertion P_i depends on the this interpretation as represented by $(x : \mathcal{I}(\tau, i-1))$. This is an application of the *HOAS* approach [36] which expedites the mechanization of $sProp_i$, as we discuss later.

To allow *higher-order* quantifications in $sProp_i$, *i.e.,* quantification over $sProp_i$, we define a special type ϕ which represents the stratified propositions *themselves*, and *stratify* the type interpretation $\mathcal{I}(\cdot, i)$ with the stratification index i. Then the interpretation of ϕ is decided by the stratification index of the interpretation: $\mathcal{I}(\phi, i) = sProp_i$. As a result, a $sProp_i$ assertion can assert quantifications over $sProp_{i-1}$ with a lower index, enabling higher-order quantifications.

Syntax and interpretation of sProp$_i$. $sProp_i$ is a *data type* that represents the syntax of assertions in Lilo where the semantic domain is *iProp*, which (roughly) has the following definition: $iProp = \Sigma \to \textbf{\textit{Prop}}$. In the definition, Σ is a (non-step-indexed) PCM and ***Prop*** is the type of meta-level (*i.e.,* Coq) propositions. The base of the stratification $sProp_0$ is an empty set $\varnothing$. For $i > 0$, $sProp_i$ includes syntax for the usual connectives of separation logic such as the separating conjunction $P_i * Q_i$. It also has syntax for various modalities, such as persistence modality $\Box P_i$ and the update modality $\Rightarrow\!\!\!\!\!\!\cdot\ P_i$ [22]. Interpretation of these usual assertions is trivial, for instance, $[\![P_i * Q_i]\!]_i = [\![P_i]\!]_i * [\![Q_i]\!]_i$.

$sProp_i$ also supports *first-order custom ghost states*, for which the user can use the assertion $\boxed{\ulcorner \alpha \urcorner}^{\gamma}$ which asserts ownership of an element α of the carrier set of a *small PCM (sPCM)* allocated at the ghost location γ. Small PCMs are PCMs that *do not* depend on $sProp_i$, thus cannot describe higher-order custom ghost states. On the other hand, because of this restriction, $sProp_i$ *can* depend on sPCMs, which is crucial for allowing custom ghost states. sPCMs have a natural embedding into PCMs which makes the interpretation of $\boxed{\ulcorner \alpha \urcorner}^{\gamma}$ trivial.

Stratification of $sProp_i$ occurs with the *higher-order* quantifiers, which means we can write an assertion of type $sProp_i$ with (universal or existential) quantification over $sProp_{i-1}$. More concretely, the quantifiers of $sProp_i$ quantify over a type of $sProp_i$ which include ϕ, the type of assertions. Because the type interpretation of ϕ at index i, $\mathcal{I}(\phi, i)$, is defined as $sProp_i$, the assertion $\forall_\phi (x : \mathcal{I}(\phi, i-1)).\, P_i$ is interpreted as $\forall (x : sProp_{i-1}).\, [\![P_i\, x]\!]_i$, which captures higher-order universal quantification (similar for the existential quantifier). This makes the logic described by $sProp_i$ a higher-order logic, and important features of Lilo such as the invariants and the obligation lists depend on the power of higher-order quantification.

An assertion P_i of $sProp_i$ can be *lifted* to an assertion of $sProp_j$ where $j > i$ using the *lift* assertion, which enables us to write assertions that include assertions of different stratification indices. A lifted assertion $\uparrow P_{i-1}$ is interpreted with the original stratification index $[\![P_{i-1}]\!]_{i-1}$.

The remaining components of $sProp_i$ are called *atoms*, which are syntax for user-defined separation logic predicates that depend on *higher-order ghost states*. For example, atoms of Lilo include the invariant predicate $\boxed{P_i}^{\nu}$, the obligation list predicate $\mathit{Obls}_{th}(\Phi)$, and the promise predicate $\xrightarrow{\kappa}\!\!\diamond P_i$. The most important feature of atoms is that the interpretation of atoms *do not* interpret the $sProp_i$s in their parameters. For example, the interpretation of the invariant predicate $\boxed{P_i}^{\nu}$ is $\boxed{P_i}^{\nu}$ (of type *iProp*), where the predicate keeps P_i *not* interpreted. This is because we implement higher-order ghost states by letting the predicates depend on the syntactic data type $sProp_i$, not the actual semantics *iProp*. Consequently, PCMs used for the definition of higher-order ghost states do not depend on *iProp*, avoiding an unsound cyclic definition.

Mechanizing and working with $sProp_i$. The definition of $sProp_i$ with its full power to allow higher-order quantifiers is not easy to mechanize in Coq, because a naive definition easily results in universe inconsistency. To implement a mechanized definition of $sProp_i$, we apply the *PHOAS* approach [5] to develop stratified syntax that allows higher-order quantifiers.

Using $sProp_i$ means every predicate in Lilo is indexed by its stratification index, and this introduces some complexity in proofs, especially when writing invariants. However, the index does not need to increase unless there is a higher-order quantification, which is not commonly required, and most examples discussed in §7 use only two indices while the most complex example (the elimination stack) uses four indices. Moreover, actual proofs in Lilo are carried out in the semantic domain by interpreting $sProp_i$ to *iProp* whenever necessary, which allows us to utilize the Iris Proof Mode [26], a powerful tool that streamlines concurrent separation logic proofs.

5.2 Invariants with Stratified Propositions

In this section, we give an overview of the invariants in Lilo. Instead of describing the model in its full detail, we focus on demonstrating how we apply stratified propositions to obtain higher-order ghost states. Our model of invariants closely follows that of Iris with slight modifications to incorporate stratified propositions, and Jung et al. [22, §7] discusses the original model in detail.

World satisfactions. Invariants in Lilo are governed by a protocol called *world satisfactions* which is a collection of predicates W_is. Concretely, the protocol is defined by a (syntactic) predicate Ws_n of type $sProp_n$ defined as $Ws_n \triangleq \mathbin{\ast}_{i<n}(\uparrow^{n-i-1} W_i)$. Here, we define $Ws_0 \triangleq \mathsf{True}$ and $\uparrow^n P_i$ lifts P_i from $sProp_i$ to $sProp_{i+n}$. Namely, Ws_n collects W_is where $0 \le i < n$.

A predicate W_i is called a *stratified world satisfaction*:

$$W_i : sProp_{i+1} \triangleq \exists_{\mathbb{N} \xrightarrow{\mathsf{fin}} \phi}(I : \mathcal{I}(\mathbb{N} \xrightarrow{\mathsf{fin}} \phi,\ i)).\ \uparrow (\mathit{WHas}_i(I) \ast \mathop{\ast}_{\nu \in \mathrm{dom}(I)} ((I(\nu) \ast \boxed{\{\nu\}}^{\gamma_D}) \vee \boxed{\{\nu\}}^{\gamma_E}))$$

$$[\![\boxed{P_i}^{\nu}]\!]_i \triangleq \boxed{\circ[\nu \mapsto \underline{ag}(P_i)]}^{\gamma_I} \quad [\![\mathit{WHas}_i(I)]\!]_i \triangleq \boxed{\bullet \underline{ag}(I)}^{\gamma_I} \ (ag \text{ is mapped pointwise over } I.)$$

Here, W_i with index i has type $sProp_{i+1}$ because it includes $\exists$ over I of type $\mathcal{I}(\mathbb{N} \xrightarrow{\mathsf{fin}} \phi,\ i)$, which is a finite map from the natural numbers to $sProp_i$ that tracks all existing invariants. The remaining part of the definition has the type $sProp_i$, and we use a lift $\uparrow$ to embed $sProp_i$ into $sProp_{i+1}$.

A stratified world satisfaction require two kinds of atoms, each representing higher-order ghost states $\mathit{WHas}_i(I)$ and $\boxed{P_i}^{\nu}$. The interpretation of these atoms utilize authoritative assertions and agreement algebra [22]. Intuitively, $\mathit{WHas}_i(I)$ keeps track of a finite map I from $\mathbb{N}$ to $sProp_i$, and $\boxed{P_i}^{\nu}$ is a persistent predicate that holds a singleton map $[\nu \mapsto ag(P_i)]$. Then, by the underlying algebra, we can derive $I(\nu) = P_i$ from $\mathit{WHas}_i(I)$ and $\boxed{P_i}^{\nu}$, which lets us retrieve P_i from W_i.

FUPD-DEF
$$ {}^{n,\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} P_n \triangleq Ws_n * \lfloor \overline{\mathcal{E}_1} \rfloor^{\gamma_E} \mathrel{-\!\!*} \dot{\Rrightarrow}(Ws_n * \lfloor \overline{\mathcal{E}_2} \rfloor^{\gamma_E} * P_n) $$

INV-CLOSE
$$ \frac{[\![P_i]\!]_i * ([\![P_i]\!]_i \ {}^{n,\mathcal{E}\setminus\{\nu\}}\!\Rrightarrow^{\mathcal{E}} \mathsf{True}) * tgt \ \precsim^{th}_{n,\mathcal{E}} \ src}{tgt \ \precsim^{th}_{n,\mathcal{E}\setminus\{\nu\}} \ src} $$

INV-ALLOC
$$ \frac{i < n \quad \nu \in \mathcal{N}}{[\![P_i]\!]_i \Rrightarrow^*_n \boxed{P_i}^{\nu}} $$

INV-OPEN
$$ \frac{\boxed{P_i}^{\nu} * ((([\![P_i]\!]_i * ([\![P_i]\!]_i \ {}^{n,\mathcal{E}\setminus\{\nu\}}\!\Rrightarrow^{\mathcal{E}} \mathsf{True})) \mathrel{-\!\!*} tgt \ \precsim^{th}_{n,\mathcal{E}\setminus\{\nu\}} \ src) \quad i < n}{tgt \ \precsim^{th}_{n,\mathcal{E}} \ src} $$

Fig. 6. Selected invariant rules of Lilo.

The rest of the definition is necessary to implement an interface that enables opening and closing of invariants, and it closely follows the construction by Jung et al. [22, §7]. We omit the details since they are irrelevant to higher-order ghost states.

Invariant rules. We present selected invariant rules of Lilo in Fig. 6. We define the *fancy update modality* with Ws_n, and we follow the notations of Iris with a slight modification to denote the stratification index n. Also, the simulation weakest precondition $tgt \ \precsim^{th}_{n,\mathcal{E}} \ src$ is roughly defined as $Ws_n * \lfloor \overline{\mathcal{E}} \rfloor^{\gamma_E} \mathrel{-\!\!*} \dot{\Rrightarrow} tgt \ \precsim^{th} \ src$ where $tgt \ \precsim^{th} \ src$ is the underlying thread-local simulation relation. Then, most of the invariant rules can be obtained by unfolding the definitions.

Note that the fancy update modality and the simulation weakest precondition are both stratified, where the index is decided by Ws_n inside each definition. This results in the $i < n$ requirement in some of the rules, such as INV-ALLOC, where i is the index of the proposition registered to the invariant and n is the index of Ws_n. This is because Ws_n only includes W_is with $i < n$, and we need W_j to retrieve a proposition with index j.

6 Generalized Rules of Lilo: Delayed Promises

The complete rules of Lilo involve *delayed promises* which we omitted in the previous sections. They enable a more natural description of *causal dependencies* between promises, which means the fulfillment of a promise depends on the fulfillment of some other promises.

For example, consider the following code executed by a thread T_2:

$$ \texttt{do}\ \{\ \mathcal{Y};\ a \coloneqq X;\ \}\ \texttt{while}\ (a \neq 1);\ \mathcal{Y};\ Y \coloneqq 2;\ \mathcal{Y};\ \texttt{ret}\ 0 $$

Thread T_2 waits for the value of a shared location X to be updated to 1 in a while loop and writes 2 to Y after it exits the loop. Assuming that there exists another thread T_1 that first writes 1 to X and waits for the value at Y to be updated to 2, the termination argument of this program in Lilo uses two promises $\overset{\kappa_1}{\longrightarrow}\!\!\diamond \boxed{X \mapsto 1}^{x}$ and $\overset{\kappa_2}{\longrightarrow}\!\!\diamond \boxed{Y \mapsto 2}^{y}$ where the first one is fulfilled by T_1 and the second one by T_2. However, the promise $\overset{\kappa_2}{\longrightarrow}\!\!\diamond \boxed{Y \mapsto 2}^{y}$ of T_2 can be fulfilled *only after* the promise $\overset{\kappa_1}{\longrightarrow}\!\!\diamond \boxed{X \mapsto 1}^{x}$ of T_1 is fulfilled, exhibiting a causal dependency between the two promises.

In this case, thread T_2 can make a *delayed promise* $\overset{\kappa_2}{\longrightarrow}\!\!\text{⧖} \boxed{Y \mapsto 2}^{y}$ instead of a real promise, and then thread T_1 can *activate* the delayed promise after it fulfills $\overset{\kappa_1}{\longrightarrow}\!\!\diamond \boxed{X \mapsto 1}^{x}$. While the promise κ_2 is delayed, T_2 does not submit progress credits for κ_2 as required by YIELD-TGT while waiting in the loop, and when the promise is activated, T_2 knows that the value at X is 1 and can exit the loop.

Our case studies show that delayed promises enable more natural liveness reasoning (§7). We discuss some core concepts here, and guide interested readers to our Coq development [2].

Rules of delayed promise. To support delayed promises, we extend the rules of Lilo by introducing $⧖_\kappa$ called the *activation token* (Fig. 7). We obtain *two* activation tokens when we allocate a new liveness obligation κ using CRED-NEW2, and we need to spend one of them when we add κ to an obligation list using OBLS-ADD2. Then, we can obtain a delayed promise $\overset{\kappa}{\longrightarrow}\!\!\text{⧖}\ P_i$ using DP-GET.

When we have a delayed promise $\overset{\kappa}{\longrightarrow}\!\!\text{⧖}\ P_i$ and an activation token $⧖_\kappa$ (the remaining one from CRED-NEW2), we can consume $⧖_\kappa$ and *activate* the delayed promise to obtain a promise $\overset{\kappa}{\longrightarrow}\!\!\diamond\ P_i$ using

ACT-PERS
$\mathsf{persistent}(\bowtie_\kappa)$

ACTIVATE
$$\frac{\boxtimes_\kappa * \boxtimes_\kappa}{\dot{\Rightarrow} \bowtie_\kappa}$$

NOT-ACT
$$\frac{\boxtimes_\kappa * \bowtie_\kappa}{\mathsf{False}}$$

CRED-NEW2
$$\frac{\ell,\ n \in \mathbb{N}}{\dot{\Rightarrow} \exists \kappa,\ \blacklozenge_\kappa \lceil \ell,\ n \rceil * \diamondsuit_\kappa(\ell,\ n) * \boxtimes_\kappa * \boxtimes_\kappa}$$

OBLS-ADD2
$$\frac{Obls_{th}(\Phi) * \mathsf{persistent}([\![P_i]\!]_i) * \diamondsuit_\kappa(1,\ 1) * \boxtimes_\kappa}{\dot{\Rightarrow} Obls_{th}((\kappa,\ P_i) :: \Phi)}$$

OBLS-FULFILL2
$$\frac{Obls_{th}(\Phi) * [\![P_i]\!]_i * \bowtie_\kappa \quad (\kappa,\ P_i) \in \Phi}{\dot{\Rightarrow} Obls_{th}(\Phi \setminus (\kappa,\ P_i))}$$

DP-PERS
$\mathsf{persistent}(\overset{\kappa}{\boxtimes} P_i)$

DP-GET
$$\frac{Obls_{th}(\Phi) \quad \Phi(\kappa) = P_i}{\overset{\kappa}{\boxtimes} P_i}$$

DP-ACT
$$\frac{\overset{\kappa}{\boxtimes} P_i * \boxtimes_\kappa}{\dot{\Rightarrow} \overset{\kappa}{\diamond} P_i}$$

PROM-DEF
$\overset{\kappa}{\diamond} P_i \triangleq \bowtie_\kappa * \overset{\kappa}{\boxtimes} P_i$

YIELD-TGT2
$$\frac{(Obls_{th}(\Phi_1 \circ \Phi_2) * \mathop{\Large *}_{\kappa \in \mathrm{dom}(\Phi_1)} \boxtimes_\kappa * €)\ {}^{n,\mathcal{E}}\!\Rrightarrow\!{}^{\top}\ k_t\ \precsim^{th}_{n,\top}\ \mathcal{Y};\ k_s}{(Obls_{th}(\Phi_1 \circ \Phi_2) * \mathop{\Large *}_{\kappa \in \mathrm{dom}(\Phi_1)} \boxtimes_\kappa * \mathop{\Large *}_{\kappa \in \mathrm{dom}(\Phi_2)} \diamondsuit_\kappa(1,\ 1)) \mathrel{-\!\!*} \mathcal{Y};\ k_t\ \precsim^{th}_{n,\mathcal{E}}\ \mathcal{Y};\ k_s}$$

Fig. 7. Selected rules of obligation list and delayed promise.

DP-ACT. By activating a delayed promise, we can use the promise progress rule PROM-PROGRESS, which does not hold for a delayed promise. Also, once we activate a promise, we can obtain a persistent token $\bowtie_\kappa$ called an *activated token* from PROM-DEF, which contradicts with $\boxtimes_\kappa$ (NOT-ACT). A thread can fulfill κ only when it is activated, as $\bowtie_\kappa$ is required by OBLS-FULFILL2.

Generalized rule for the yield. When a promise is delayed, *i.e.,* when we have an activation token $\boxtimes_\kappa$, we can submit it *instead* of progress credits $\diamondsuit_\kappa(1,\ 1)$ when executing a $\mathcal{Y}$. Then the activation token is returned after the thread returns from the yield. This principle is captured by YIELD-TGT2, which decomposes the obligation list into delayed ones Φ_1 and activated ones Φ_2 and requires $\boxtimes_\kappa$ for Φ_1 and $\diamondsuit_\kappa(1,\ 1)$ for Φ_2. Then the rule returns $\boxtimes_\kappa$ for Φ_1 and a scheduler credit €.

The YIELD-TGT2 rule is also relaxed compared to YIELD-TGT since it requires a *view shift* from $\mathcal{E}$ to $\top$ instead of directly requiring the $\top$ mask. This allows activation tokens to be *shared* via invariants, which is necessary for other threads to activate corresponding delayed promises.

7 Case Studies

We present several verification results carried out with Lilo. Results presented in this section are simplified for the sake of space, such as the specifications or discussion about the proofs. Especially, we omit details regarding *sProp*s and write them as if they were *iProp*s. Our Coq mechanization [2] contains precise and complete results.

7.1 Termination Guaranteeing Specifications for Locks

View shift and spinlock. We present a useful lemma for the spinlock spec.

$$\frac{\mathsf{isSL}(\kappa_S,\ x,\ L,\ N) * \diamondsuit_{\kappa_S}(\ell',\ n') * \blacklozenge_{\kappa_{U'}} \lceil \ell',\ n' \rceil * \boxtimes_{\kappa_{U'}} * \overset{\kappa_{U'}}{\boxtimes} \boxed{shot}^{\gamma_{U'}} * \boxed{pend_1}^{\gamma_{U'}}}{locked(\kappa_U) * Obls_i((\kappa_U,\ \boxed{shot}^{\gamma_U}) :: \Phi) \Rrightarrow_\top locked(\kappa_{U'}) * Obls_i(\Phi) * \boxed{shot}^{\gamma_U} * \bowtie_{\kappa_{U'}}} \quad \text{(SL-PASS)}$$

SL-PASS allows "passing" the unlock obligation between threads through a *view shift*. This rule is necessary when the thread acquiring the lock differs from the one releasing it. It enables the acquiring thread to fulfill its unlock obligation κ_U by passing the unlock obligation to another thread, which has made a delayed promise $\kappa_{U'}$ to unlock the lock. To prevent this lock-passing from happening indefinitely, where no thread actually unlocks the lock, the lemma requires progress credits for the finite delay obligation κ_S.

Ticket lock. Here is the code and a simplified spec for Ticketlock that guarantees termination.

$$\texttt{def lock}_{\texttt{tk}}(n,\, o) \;=\; \mathcal{Y};\, tk \coloneqq \texttt{FAI}(n);\, \texttt{do}\, \{\, \mathcal{Y};\, own \coloneqq o;\, \mathcal{Y};\, \}\, \texttt{while}\, (tk \neq own);\, \mathcal{Y}; \qquad \text{(Ticketlock)}$$

$$\texttt{def unlock}_{\texttt{tk}}(n,\, o) \;=\; \mathcal{Y};\, own \coloneqq o;\, \mathcal{Y};\, o \coloneqq own + 1;\, \mathcal{Y};$$

$$\{\mathit{Obls}_{th}([])\}\ \texttt{lock}_{\texttt{tk}}(n,\, o)\ \{\exists \kappa_U.\, \mathit{Obls}_{th}([(\kappa_U, \boxed{\mathit{shot}})]) * \Diamond_{\kappa_U}(L,\, 1) * \mathit{locked}(\kappa_U)\}_{\top}$$

$$\{\mathit{Obls}_{th}([(\kappa_U, \boxed{\mathit{shot}})]) * \Diamond_{\kappa_U}(1,\, 3) * \mathit{locked}(\kappa_U)\}\ \texttt{unlock}_{\texttt{tk}}(n,\, o)\ \{\mathit{Obls}_{th}([])\}_{\top}$$

As discussed in §4.3, this specification does not require a finite delay obligation because Ticketlock is starvation-free, unlike a spinlock. Starvation-freedom ensures that threads will eventually acquire the lock if it is always eventually unlocked, and we only need an unlock obligation κ_U.

However, proving Ticketlock requires reasoning about a non-local linearization point where the unlocking thread "locks" the lock for the next waiting thread at $o \coloneqq own + 1$. To reason about this, the proof leverages delayed promises. When a thread requests a ticket, it also makes a delayed promise to eventually unlock, and the current lock holder activates this promise when unlocking, designating the next thread as the new lock owner.

7.2 Termination Guaranteeing Specification for Concurrent Stacks

Stack specification. We prove a termination guaranteeing specification of Treiber's stack [46] and the elimination stack [15]. In particular, we prove a mixture of logically atomic triples (LATs) [8, 24, 42] and HOCAP [43] style logical atomicity.[4] For example, the triple for a push operation of a stack with an empty obligation list is given below (ghost names omitted) :

$$\begin{aligned}\text{isStack}(\kappa, s) \vdash \{&\text{AU}(\lambda\, St.\, \texttt{Stack}(St),\, \lambda\, St.\, \texttt{Stack}(v :: St),\, R) * \mathit{Obls}_{th}([]) \;*\; \Diamond_{\kappa}(1,\, 1)\} \qquad \text{(HT-ST)}\\ &\texttt{push}(s, v)\ \{R * \mathit{Obls}_{th}([])\}_{\top}\end{aligned}$$

Here, $\text{AU}(P, Q, R)$ is an "atomic update" [19, 23], which represents the obligation to transform P into Q at some atomic instruction, while returning R. AU in the precondition allows for a client to *open invariants* around push (similar to INV-OPEN), even though it is not an atomic operation.

Treiber stack. Treiber's stack is a concurrent, lock-free linked list with operations $\texttt{push}(s, v)$ that adds v to the head of the stack s, and $\texttt{try\_pop}(s)$ that removes the element at the head of the stack, or returns $\bot$ if the stack is empty. In push and try_pop, the operating threads will perform a loop that **(1)** reads and keeps a snapshot of the current head node; **(2)** attempts to update the head node to a new node with a **CAS** operation. This **CAS**-loop pattern is similar to that of a spinlock: a failed **CAS** failed means that some other thread has succeeded in its operation and updated the head node. Thus, similar to a spinlock, we can guarantee termination of all stack operations by setting up a liveness obligation that obligates finite usage of the stack.

Elimination stack. Elimination stack [15] is an enhancement of the Treiber stack that improves scalability by allowing a pair of push and pop that failed its CAS to "eliminate" each other and succeed. The elimination stack is a principle example of *non-local linearization point* called helping: the linearization point of one operation (push) happens in another (a matched try_pop). In Lilo, reasoning about helping is carried out by sharing atomic updates between threads, as in Iris. A push thread will put its atomic update inside an invariant, and a try_pop thread will obtain this atomic update and resolve it in place of the push thread.

Remark. We also note that our proofs are natural extensions of existing safety proofs [18]. We simply imported the safety invariants and extended them by adding liveness obligations and progress credits. We expect that porting additional lock-free data structures can be done similarly.

[4] Proving the LAT for the elimination stack requires putting the Hoare triple in an invariant, which is currently not possible in *sProp*. Iris solves this with nontrivial constructions [19, 41]. We leave proving LAT in *sProp* as a future work.

Message Passing with stacks. We test our spec by proving the following:

$$\mathcal{Y};\ \texttt{skip};\ \mathcal{Y};\ \texttt{push}(s,1);\ \ \Big\|\ \ \textbf{do}\ \{\ \mathcal{Y};\ v \coloneqq \texttt{try\_pop}(s);\ \mathcal{Y};\ \}\ \textbf{while}\ (v = \bot);\ \mathcal{Y};\ \textbf{ret}\ v \qquad \text{(STACK-MP)}$$

Under fair scheduling, the program terminates. This is because the first thread will eventually be scheduled and attempt to push a value to the stack with a **CAS** instruction. This CAS must succeed, since the second thread only does a pop operation, which does not modify the stack state. For the second thread, it must eventually pop the pushed value by the first thread, and return with a value 1. In essence, we exploit the fact the empty pops does not interfere with push operations.

7.3 Client Patterns

Infinite message passing. We prove that INF-MP (§2.1) refines its spec.

$$\begin{array}{l}\textbf{while}\ (1)\ \{\ \mathcal{Y};\ X \coloneqq 1;\\ \quad\textbf{do}\ \{\ \mathcal{Y};\ a \coloneqq X;\ \}\ \textbf{while}\ (a = 1);\ \mathcal{Y};\ \mathit{print}(a);\ \}\end{array}\ \Big\|\ \begin{array}{l}\textbf{while}\ (1)\ \{\ \mathcal{Y};\ X \coloneqq 2;\\ \quad\textbf{do}\ \{\ \mathcal{Y};\ b \coloneqq X;\ \}\ \textbf{while}\ (b = 2);\ \mathcal{Y};\ \mathit{print}(b);\ \}\end{array}$$

$$\textbf{while}\ (1)\ \{\ \mathcal{Y};\ \mathit{print}(2);\ \}\ \Big\|\ \textbf{while}\ (1)\ \{\ \mathcal{Y};\ \mathit{print}(1);\ \} \qquad \text{(INF-MP-SPEC)}$$

The invariant encodes a two-state transition system that switches when a thread sends a message by updating X, and it is designed to let a thread earn progress credits to do induction or find out that it can exit the loop and update X. Delayed promises are used in this example to reduce the complexity of the invariant, enabling an intuitive reasoning. Specifically, when a thread updates X, changes the ghost state, and starts a new loop, a promise from the other thread to update X again is required to do induction (using PROM-PROGRESS and CRED-IND) to exit the loop. To carry out this reasoning, a thread can create a delayed promise at the instant of updating X, and the other thread can activate it at the next instant of updating X. We remark that Lilo supports coinductive reasoning through a coinduction lemma for $\precsim^{th}_{\mathcal{E}}$, which utilizes FreeSim to facilitate the proof [6].

Spinlock passing. We prove that the 'lock passing' program (D'Osualdo et al. [10, §2.1]) terminates.

$$\mathcal{Y};\ \texttt{lock}(x);\ \mathcal{Y};\ D \coloneqq 1;\ \mathcal{Y};\ \textbf{ret}\ 0\ \Big\|\ \mathcal{Y};\ \textbf{do}\ \{\ \mathcal{Y};\ d \coloneqq D;\ \}\ \textbf{while}\ (d \neq 1);\ \mathcal{Y};\ \texttt{unlock}(x);\ \mathcal{Y};\ \textbf{ret}\ 0 \qquad \text{(LP)}$$

The proof of this program uses SL-PASS (§7.1). When thread 1 writes 1 to D, it simultaneously passes the unlock obligation to thread 2, activating thread 2's delayed promise to unlock the lock. Then thread 2 proceeds to unlock the lock.

Scheduler non-determinism. We prove the termination of SCH-ND presented in §2.1.

$$\textbf{while}\ (d = 0)\ \{\ \mathcal{Y};\ \texttt{lock}(x);\ \mathcal{Y};\ d \coloneqq D;\ \mathcal{Y};\ \texttt{unlock}(x);\ \};\ \mathcal{Y};\ \textbf{ret}\ 0\ \Big\|\ \mathcal{Y};\ D \coloneqq 1;\ \mathcal{Y};\ \textbf{ret}\ 0 \qquad \text{(SCH-ND)}$$

High level reasoning and the invariant of this proof are similar to the one introduced in MP. The only difference is that there is a lock/unlock pair wrapping the read operation. Thus the only challenge in proving termination of thread 1 is to 'replenish' the progress credits required to lock/unlock, and this is enabled by relating the finite delay obligation κ_S of the spinlock with the obligation of thread 2 to write. With a promise from thread 2, thread 1 can exit the loop, or obtain a progress credit and restore the initial state of the inductive hypothesis.

8 Related Work and Discussion

Concurrent separation logic. Concurrent separation logics [4, 34] (CSLs) are oriented toward verifying properties of concurrent programs in a thread-local way. Modern CSLs, notably Iris [22, 24] and its relatives achieve a high level of abstraction for reasoning about various safety properties [9, 20, 27, 33, 35, 38]. Additionally, there exists several relational separation logics for proving refinements [12, 13, 44, 45]. However, unlike Lilo, they do not develop high-level abstractions that support liveness reasoning.

Separation logic for liveness. We described the most closely related work to Lilo in the background (§2). We now discuss how Lilo relates to some other existing separation logic for liveness.

Reinhard et al. [37] propose a separation logic for verifying termination of programs that abruptly terminate. They develop predicates called "obligation" and "credit", which are only superficially related to Lilo. For instance, their notion of a credit is a token to allow a thread to busy-wait. More importantly, their rule for a loop forbids the looping thread to hold any obligations, which severely limits how threads can depend on liveness of other threads. In fact, allowing threads to depend on other threads' liveness is an important challenge for a logic for liveness because it needs to prevent circularity, and TaDA Live [10], LiLi [30, 31] and Lilo all put significant effort into this.

Timany et al. develop a CSL *Fairis* using Trillium [45], that is geared toward proving liveness of concurrent programs under fair scheduling via refinement. Although Lilo and Fairis shares a similar verification goal, Lilo's advantage lies in its novel abstractions that enable modular liveness reasoning. In addition, Lilo develops modular specifications that guarantee termination and address blocking and delay. These high-level abstractions are not developed in Fairis.

Lilo's notion of obligation list is inspired by the notion of obligations in TaDA Live [10]. Both of them are designed to support liveness reasoning in a thread-local way and represent some notion of obligation that must be fulfilled. However, their meta-level mechanism is completely different. Specifically, TaDA Live adopts a classical interpretation of the separating conjunction, which means $P * Q \Rrightarrow P$, and this is crucial for the soundness of TaDA Live. In contrast, Lilo is based on Iris-style resource algebra (without step-indexing) which leads to an affine logic that allows $P * Q \Rightarrow P$, and the soundness of Lilo is based on the soundness of FOS. We believe this makes Lilo a favorable approach to adapt to existing frameworks that utilize Iris-style resource algebras.

Credits as a resource. Lilo's progress credits is influenced by existing work that represent credits as a resource, in particular Mével et al. [33]'s time credits for reasoning about time complexity, and Spies et al. [40]'s transfinite time credits for proving termination. However, both of them target sequential programs and do not have a good support for concurrent programs.

Higher-order ghost states. To the best of our knowledge, there are currently two approaches to higher-order ghost states in separation logic. One approach is to utilize step-indexing, as done by Iris [21], which is known to be ill-suited for liveness reasoning as discussed in §5. Moreover, it is known that step-indexing fundamental conflicts with modular transitivity [3, 16, 17], making it inadequate for frameworks such as CCR [39].

Another approach is to separate the syntax of the logical assertions from its semantic interpretation, pioneered by Nola [32]. Nola shows that it is possible to support Iris-style invariants and borrows without step-indexing, and Matsushita applies Nola to the verification of type systems, such as Rust-style borrows. We adapt this approach and develop stratified propositions to enable higher-order reasoning in Lilo without step-indexing, and apply it to verify liveness of concurrent programs. We believe Nola-style approach and our stratified propositions are applicable to other non-step-indexed frameworks, such as Simuliris [14] and CCR [39].

However, we remark that stratified propositions currently does not support *impredicative* features of Iris. Consequently, as discussed in §7.2, proving the LAT for the elimination stack is currently not possible. We believe this is because the LAT requires impredicative features, *e.g.*, putting a predicate of index $i + 1$ in an invariant of index i.

Limitations and future work. As discussed in §4.3, Lilo currently does not support specifications for functions involving unbounded dynamic nesting of blocking functions. Also, Lilo's stratified propositions currently does not support Iris's LATs. We leave a deeper investigation to these limitations as future work.

Acknowledgments

This work was supported in part by Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-IT2102-03, and by a National Research Foundation of Korea (NRF) grant funded by the Ministry of Science and ICT (MSIT) of the Korea government (No. RS-2024-00347786). Chung-Kil Hur is the corresponding author.

Data Availability Statement

The Coq formalization of this paper is available on Zenodo [29].

References

[1] Bowen Alpern and Fred B Schneider. 1987. Recognizing safety and liveness. *Distributed computing* 2, 3 (1987), 117–126.

[2] Anonymous Author(s). 2024. Lilo: Coq development (supplementary material).

[3] Nick Benton and Chung-Kil Hur. 2010. Step-Indexing: The Good, the Bad and the Ugly. In *Modelling, Controlling and Reasoning About State, Proceedings of Dagstuhl Seminar 10351* (modelling, controlling and reasoning about state, proceedings of dagstuhl seminar 10351 ed.). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany. https://www.microsoft.com/en-us/research/publication/step-indexing-the-good-the-bad-and-the-ugly/ An earlier version of this work was presented at the Workshop on Syntax and Semantics of Low-Level Languages (LOLA) in July 2010..

[4] Stephen Brookes. 2007. A semantics for concurrent separation logic. *Theoretical Computer Science* 375, 1-3 (2007), 227–270.

[5] Adam Chlipala. 2008. Parametric higher-order abstract syntax for mechanized semantics. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming* (Victoria, BC, Canada) *(ICFP '08)*. Association for Computing Machinery, New York, NY, USA, 143–156. doi:10.1145/1411204.1411226

[6] Minki Cho, Youngju Song, Dongjae Lee, Lennard Gäher, and Derek Dreyer. 2023. Stuttering for Free. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 281 (oct 2023), 28 pages. doi:10.1145/3622857

[7] Pedro da Rocha Pinto. 2016. *Reasoning with time and data abstractions*. Ph. D. Dissertation. Imperial College London, UK.

[8] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *ECOOP 2014 – Object-Oriented Programming*, Richard Jones (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 207–231.

[9] Hoang-Hai Dang, Jaehwang Jung, Jaemin Choi, Duc-Than Nguyen, William Mansky, Jeehoon Kang, and Derek Dreyer. 2022. Compass: strong and compositional library specifications in relaxed memory separation logic. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 792–808. doi:10.1145/3519939.3523451

[10] Emanuele D'Osualdo, Julian Sutherland, Azadeh Farzan, and Philippa Gardner. 2021. TaDA Live: Compositional Reasoning for Termination of Fine-Grained Concurrent Programs. *ACM Trans. Program. Lang. Syst.* 43, 4, Article 16 (nov 2021), 134 pages. doi:10.1145/3477082

[11] Cormac Flanagan and Stephen N. Freund. 2020. The anchor verifier for blocking and non-blocking concurrent software. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 156 (Nov. 2020), 29 pages. doi:10.1145/3428224

[12] Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2018. ReLoC: A Mechanised Relational Logic for Fine-Grained Concurrency. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (Oxford, United Kingdom) *(LICS '18)*. Association for Computing Machinery, New York, NY, USA, 442–451. doi:10.1145/3209108.3209174

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

[14] Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. 2022. Simuliris: A Separation Logic Framework for Verifying Concurrent Program Optimizations. *Proc. ACM Program. Lang.* 6, POPL, Article 28 (jan 2022), 31 pages. doi:10.1145/3498689

[15] Danny Hendler, Nir Shavit, and Lena Yerushalmi. 2004. A Scalable Lock-Free Stack Algorithm. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures* (Barcelona, Spain) *(SPAA '04)*. Association for Computing Machinery, New York, NY, USA, 206–215. doi:10.1145/1007912.1007944

[16] Chung-Kil Hur and Derek Dreyer. 2011. A kripke logical relation between ML and assembly. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 133–146. doi:10.1145/1926385.1926402

[17] Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. 2012. The marriage of bisimulations and Kripke logical relations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. Association for Computing Machinery, New York, NY, USA, 59–72. doi:10.1145/2103656.2103666

[18] Iris Team. 2024. Iris examples. https://gitlab.mpi-sws.org/iris/examples

[19] Ralf Jung. 2019. Logical Atomicity in Iris: the Good, the Bad, and the Ugly. Iris Workshop. https://people.mpi-sws.org/~jung/iris/talk-iris2019.pdf

[20] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (dec 2017), 34 pages. doi:10.1145/3158154

[21] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-Order Ghost State. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) *(ICFP 2016)*. Association for Computing Machinery, New York, NY, USA, 256–269. doi:10.1145/2951913.2951943

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

[23] Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2019. The Future is Ours: Prophecy Variables in Separation Logic. *Proc. ACM Program. Lang.* 4, POPL, Article 45 (dec 2019), 32 pages. doi:10.1145/3371113

[24] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. *ACM SIGPLAN Notices* 50, 1 (2015), 637–650.

[25] Bernhard Kragl and Shaz Qadeer. 2021. The Civl Verifier. In *2021 Formal Methods in Computer Aided Design (FMCAD)*. 143–152. doi:10.34727/2021/isbn.978-3-85448-046-4_23

[26] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive Proofs in Higher-Order Concurrent Separation Logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 205–217. doi:10.1145/3009837.3009855

[27] Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. 2020. Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems. In *Programming Languages and Systems*, Peter Müller (Ed.). Springer International Publishing, Cham, 336–365.

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

[29] Dongjae Lee, Janggun Lee, Taeyoung Yoon, Minki Cho, Jeehoon Kang, and Chung-Kil Hur. 2025. *Artifact for Lilo: A Higher-Order, Relational Concurrent Separation Logic for Liveness.* doi:10.5281/zenodo.14927742

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

[31] Hongjin Liang and Xinyu Feng. 2017. Progress of concurrent objects with partial methods. *Proc. ACM Program. Lang.* 2, POPL, Article 20 (Dec. 2017), 31 pages. doi:10.1145/3158108

[32] Yusuke Matsushita. 2023. *Non-Step-Indexed Separation Logic with Invariants and Rust-Style Borrows.* Ph. D. Dissertation. University of Tokyo.

[33] Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2019. Time Credits and Time Receipts in Iris. In *Programming Languages and Systems*, Luís Caires (Ed.). Springer International Publishing, Cham, 3–29.

[34] Peter W. OHearn. 2007. Resources, Concurrency, and Local Reasoning. *Theor. Comput. Sci.* 375, 1–3 (apr 2007), 271–307. doi:10.1016/j.tcs.2006.12.035

[35] Sunho Park, Jaewoo Kim, Ike Mulder, Jaehwang Jung, Janggun Lee, Robbert Krebbers, and Jeehoon Kang. 2024. A Proof Recipe for Linearizability in Relaxed Memory Separation Logic. *Proc. ACM Program. Lang.* 8, PLDI, Article 154 (jun 2024), 24 pages. doi:10.1145/3656384

[36] F. Pfenning and C. Elliott. 1988. Higher-order abstract syntax. *SIGPLAN Not.* 23, 7 (jun 1988), 199–208. doi:10.1145/960116.54010

[37] Tobias Reinhard, Amin Timany, and Bart Jacobs. 2020. A separation logic to verify termination of busy-waiting for abrupt program exit. In *Proceedings of the 22nd ACM SIGPLAN International Workshop on Formal Techniques for Java-Like Programs* (Virtual, USA) *(FTfJP '20)*. Association for Computing Machinery, New York, NY, USA, 26–32. doi:10.1145/3427761.3428345

[38] Upamanyu Sharma, Ralf Jung, Joseph Tassarotti, Frans Kaashoek, and Nickolai Zeldovich. 2023. Grove: a Separation-Logic Library for Verifying Distributed Systems. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 113–129. doi:10.1145/3600006.3613172

[39] Youngju Song, Minki Cho, Dongjae Lee, Chung-Kil Hur, Michael Sammler, and Derek Dreyer. 2023. Conditional Contextual Refinement. *Proc. ACM Program. Lang.* 7, POPL, Article 39 (jan 2023), 31 pages. doi:10.1145/3571232

[40] Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2021. Transfinite Iris: resolving an existential dilemma of step-indexed separation logic. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 80–95. doi:10.1145/3453483.3454031

[41] Simon Spies, Lennard Gäher, Joseph Tassarotti, Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2022. Later credits: resourceful reasoning for the later modality. *Proc. ACM Program. Lang.* 6, ICFP, Article 100 (aug 2022), 29 pages. doi:10.1145/3547631

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

[43] Kasper Svendsen, Lars Birkedal, and Matthew J. Parkinson. 2013. Modular Reasoning about Separation for Concurrent Data Structures. In *Proceedings of ESOP* (proceedings of esop ed.). https://www.microsoft.com/en-us/research/publication/modular-reasoning-about-separation-for-concurrent-data-structures/

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

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

[46] R. K. Treiber. 1986. Systems programming: coping with parallelism.

Received 2024-10-16; accepted 2025-02-18