Later Credits: Resourceful Reasoning for the Later Modality

SIMON SPIES, MPI-SWS, Saarland Informatics Campus, Germany
LENNARD GÄHER, MPI-SWS, Saarland Informatics Campus, Germany
JOSEPH TASSAROTTI, New York University, USA
RALF JUNG, MIT CSAIL, USA
ROBBERT KREBBERS, Radboud University Nijmegen, The Netherlands
LARS BIRKEDAL, Aarhus University, Denmark
DEREK DREYER, MPI-SWS, Saarland Informatics Campus, Germany

In the past two decades, step-indexed logical relations and separation logics have both come to play a major role in semantics and verification research. More recently, they have been married together in the form of *step-indexed separation logics* like VST, iCAP, and Iris, which provide powerful tools for (among other things) building semantic models of richly typed languages like Rust. In these logics, propositions are given semantics using a step-indexed model, and step-indexed reasoning is reflected into the logic through the so-called "later" modality. On the one hand, this modality provides an elegant, high-level account of step-indexed reasoning; on the other hand, when used in sufficiently sophisticated ways, it can become a nuisance, turning perfectly natural proof strategies into dead ends.

In this work, we introduce *later credits*, a new technique for escaping later-modality quagmires. By leveraging the second ancestor of these logics—separation logic—later credits turn "the right to eliminate a later" into an ownable resource, which is subject to all the traditional modular reasoning principles of separation logic. We develop the theory of later credits in the context of Iris, and present several challenging examples of proofs and proof patterns which were previously not possible in Iris but are now possible due to later credits.

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

Additional Key Words and Phrases: Separation logic, Iris, step-indexing, later modality, transfinite

ACM Reference Format:
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 (August 2022), 29 pages. https://doi.org/10.1145/3547631

1 INTRODUCTION

In the past two decades, *step-indexed logical relations* and *separation logics* have both come to play a major role in semantics and verification research. Step-indexed logical relations, developed originally as part of the Foundational Proof-Carrying Code project [Appel and McAllester 2001; Ahmed et al. 2010], have since become an indispensable tool for building semantic models of modern type systems—such as those of Scala [Giarrusso et al. 2020], Rust [Jung et al. 2018a], and session-typed languages [Hinrichsen et al. 2021]—which include "cyclic features" like recursive

Authors' addresses: Simon Spies, MPI-SWS, Saarland Informatics Campus, Germany, spies@mpi-sws.org; Lennard Gäher, MPI-SWS, Saarland Informatics Campus, Germany, gaeher@mpi-sws.org; Joseph Tassarotti, New York University, USA, jt4767@nyu.edu; Ralf Jung, MIT CSAIL, USA, research@ralfj.de; Robbert Krebbers, Radboud University Nijmegen, The Netherlands, mail@robbertkrebbers.nl; Lars Birkedal, Aarhus University, Denmark, birkedal@cs.au.dk; Derek Dreyer, MPI-SWS, Saarland Informatics Campus, Germany, dreyer@mpi-sws.org.

2475-1421/2022/8-ART100
https://doi.org/10.1145/3547631

subtyping and higher-order state [Ahmed 2004; Birkedal et al. 2011]. Separation logic [O'Hearn et al. 2001; Reynolds 2002], though aimed originally at verifying sequential, pointer-manipulating programs, has grown into an entire subfield of program verification, spawning numerous variants—*separation logics* (plural)—which have extended it to support a wide range of challenging features, notably concurrency [O'Hearn 2007; Brookes 2007].

In recent years, these two independent developments have been married together in the form of *step-indexed separation logics*—separation logics, such as VST [Cao et al. 2018], iCAP [Svendsen and Birkedal 2014], and Iris [Jung et al. 2015, 2016; Krebbers et al. 2017a; Jung et al. 2018b], in which propositions are given semantics using a step-indexed model. Step-indexed separation logics enrich traditional separation logic with new and powerful mechanisms like *first-class storable locks* [Buisse et al. 2011], *impredicative invariants* [Svendsen and Birkedal 2014], and *higher-order ghost state* [Jung et al. 2016], mechanisms which have no precedent in prior work because they fundamentally depend on the integration of step-indexing and separation logic. These new mechanisms have proven useful in verifying correctness of fine-grained concurrent data structures [Jung et al. 2015], building semantic models of higher-order, imperative, and concurrent languages [Jung et al. 2018a; Dang et al. 2020; Hinrichsen et al. 2021], and deriving custom program logics for a variety of application domains [Hinrichsen et al. 2020; Krogh-Jespersen et al. 2020; Chajed et al. 2019; Zhang et al. 2021].

In this paper, we focus on an important and unsung feature of step-indexed separation logics: the *"later" ($\triangleright$) modality* [Appel et al. 2007]. Though glossed over in much of the literature as an esoteric technical detail best left to the grizzled experts, the later modality is in fact central to how step-indexed separation logics work, as it makes it possible to do step-indexed reasoning at a higher level of abstraction. Specifically, propositions P in step-indexed logics are interpreted as predicates over a step-index n (intuitively: "P holds true for n steps of computation"), and $\triangleright P$ is defined to be true at step-index n if P is true at step-index $n - 1$ (*i.e.,* $\triangleright P$ means that P will hold *later*—after one step of computation). As such, the later modality provides a high-level way of formalizing step-indexed arguments without being forced to reason about step-indices directly and engage in tedious "step-index arithmetic" as in earlier formulations of step-indexing [Dreyer et al. 2011].

However, in practice, the later modality is often viewed as a "necessary evil". Laters typically pop up in hypotheses when one unfolds an implicitly recursive construction (such as the impredicative invariants mentioned above), and for good reason: the laters serve as "guards" preventing paradoxes of circular reasoning [Jung et al. 2018b, §3.3, §8.2]. But once $\triangleright P$ appears in a hypothesis, the name of the game is figuring out how to *eliminate* the guarding "$\triangleright$" in order to make use of the proposition P.

This brings us to our main topic: *the later elimination problem.* Although there exist a number of techniques for eliminating laters in step-indexed proofs, there are several known situations where none of these techniques apply, thus ruling natural proof strategies out of consideration and in some cases making it unclear how to carry out the proof at all. In this paper, we propose a new technique for escaping these unfortunate situations by exploiting the fact that we are working in a separation logic. Specifically, *we treat "the right to eliminate a later" as an ownable resource* and then apply standard separation-logic reasoning to that resource. We realize this idea through a new logical mechanism we call *later credits*, and we demonstrate its effectiveness on a range of interesting use cases. But before we explain how later credits work and where they shine, let us begin by illustrating the later elimination problem with a concrete example.

The later elimination problem. To illustrate the problem, we first have to understand both the motivation for step-indexing and its limitations. We explain both with a concrete example: impredicative invariants in the step-indexed separation logic Iris [Jung et al. 2018b]. In Iris, invariants $\boxed{R}$ are used to share state between program threads. For example, we can pick $R \triangleq \exists n : \mathbb{N}.\, \ell \mapsto n$ to share access to the location ℓ and, at the same time, constrain ℓ to only store natural numbers.

Once the invariant $\boxed{R}$ is established, we can use it (called "*opening* it") by applying Iris's invariant opening rule. This rule is central enough to Iris that it was presented on page 1 of the original "Iris 1.0" paper [Jung et al. 2015], albeit in the following, "simplified for presentation purposes" form (and we will return to what is simplified about it in a moment):

$$\frac{\{R * P\}\; e\; \{R * Q\} \qquad e \text{ physically atomic}}{\boxed{R} \vdash \{P\}\; e\; \{Q\}}$$

The rule says that if we open the invariant $\boxed{R}$, then we can assume R in our precondition and have to show R holds again after evaluating e. Importantly, the rule is restricted to *atomic* expressions (*i.e.,* expressions that only take a single step). Without this restriction, other threads that are interleaved with e could potentially observe inconsistent states in which R does not hold.

What makes these invariants *impredicative* is that the R in $\boxed{R}$ can be an arbitrary Iris proposition: it can for example include Hoare triples or other impredicative invariant assertions. Impredicativity makes these invariants quite powerful: they enable reasoning about higher-order stateful programs (*i.e.,* programs storing *functions* in memory) and defining logical relations in step-indexed separation logic [Frumin et al. 2018, 2021]. Unfortunately, the price of their power is that their model is cyclic—cyclic to the extent that naive models of $\boxed{R}$ are not well-founded (*i.e.,* inductive or co-inductive definitions do not suffice). To obtain a well-founded model of $\boxed{R}$, the only known approach is to stratify the cyclic construction using step-indexing [Svendsen and Birkedal 2014; Jung et al. 2018b].

However, as explained above, a side effect of using step-indexing to resolve cycles in the model of invariants is that, when invariants are used (*i.e.,* opened), it is necessary to introduce a *later modality*, which acts as a "guard" to protect against paradoxically circular reasoning. In particular, the invariant opening rule presented above is an oversimplification; Iris's real invariant opening rule is the following:

$$\frac{\{\triangleright R * P\}\; e\; \{v.\; \triangleright R * Q\}_{\mathcal{E}\setminus\mathcal{N}} \qquad \mathcal{N} \subseteq \mathcal{E} \qquad e \text{ physically atomic}}{\boxed{R}^{\mathcal{N}} \vdash \{P\}\; e\; \{v.\, Q\}_{\mathcal{E}}}$$

Most of these additional details are not relevant to our discussion. First of all, note that the Hoare triples now bind a return value v in the postcondition Q: this is simply to allow Q to talk about the result of evaluating e. Second, note that invariant assertions are now annotated with a *namespace* $\mathcal{N}$, and Hoare triples with a *mask* $\mathcal{E}$. These mechanisms are needed to keep track of which invariants are currently open or closed, and to avoid reentrancy (*i.e.,* opening the same invariant twice while reasoning about a single step of computation). We will return to namespaces and masks in §2, but they are not our main focus in the present discussion.

Third and most importantly, note the two occurrences of the later modality ($\triangleright$) in the pre and post of the premise, which are an artifact of the step-indexed model of impredicative invariants. After applying this rule, the user needs to *eliminate* the "$\triangleright$" guarding R in the pre, so that they can use R in verifying e. Toward this end, Iris presently offers three options:

(1) *Timeless propositions.* For the subclass of so-called *timeless* propositions—which include propositions that are pure (*e.g.,* $\mathsf{even}(n)$) or assert only first-order ownership (*e.g.,* $\ell \mapsto 42$)—laters can be eliminated because the model of these propositions ignores the step-index.
(2) *Commuting rules.* The later modality commutes with most logical connectives (*e.g.,* existential quantification and separating conjunction). Thus, in many cases, we can use commuting rules to move the later out of the way.

(3) *Program steps.* With every program step, we can eliminate a guarding later. More precisely, if P is guarded by a later before the step, then the later can be removed *after* the step. (This corresponds to the intuition that $\triangleright P$ means "P will hold after one step of computation".)

The problem is that there are cases where none of these techniques apply. We illustrate such a case with an example: *nested invariants.*[1] Consider the invariant:

$$\boxed{\exists \ell.\, \boxed{\exists n : \mathbb{N}.\, \ell \mapsto n}^{\mathcal{N}_1} * \gamma \mapsto_{\text{ghost}} \ell}^{\mathcal{N}_2}$$

Here, the location ℓ is existentially quantified and connected to a logical identifier γ through a "ghost link" $\gamma \mapsto_{\text{ghost}} \ell$ (a piece of "ghost state" which is not present in the program, but useful for its verification). If we need the contents of the inner invariant $(\exists n : \mathbb{N}.\, \ell \mapsto n)$ to justify the next step, then we are in a quandary. If we open the outer invariant, we get $\triangleright \left(\exists \ell.\, \boxed{\exists n : \mathbb{N}.\, \ell \mapsto n}^{\mathcal{N}_1} * \gamma \mapsto_{\text{ghost}} \ell\right)$. After applying commuting and timelessness rules, and eliminating the existential, we are left with $\triangleright \boxed{\exists n : \mathbb{N}.\, \ell \mapsto n}^{\mathcal{N}_1}$ and $\gamma \mapsto_{\text{ghost}} \ell$. At this point, we have a later guarding $\boxed{\exists n : \mathbb{N}.\, \ell \mapsto n}^{\mathcal{N}_1}$ and are therefore stuck: invariants are not timeless (eliminating the first option), there is nothing to commute (eliminating the second option), and we need $\ell \mapsto n$ *before* (*i.e.,* as a precondition for verifying) the next step (eliminating the third option).

As we will see in this paper, the case of nested invariants is not an isolated one. There are a number of realistic scenarios in step-indexed separation logics—not common scenarios exactly, but ones that do occur periodically—where none of the "standard" later elimination options apply. At present, the only way of handling such scenarios is to attempt some non-trivial, non-local refactoring of the proof structure—or to admit defeat.

Later credits. In this paper, we present a fourth option for later elimination—*later credits.* Later credits support what we call *amortized step-indexed reasoning*—eliminating laters based on *previous* program steps. The basic idea of amortized reasoning is that we decouple the proof steps where laters are eliminated from the proof steps where we execute the program. Instead of eliminating one later after every program step (option 3 above), we obtain a credit £1 after every program step—a *later credit.* This credit can subsequently be used anywhere in the rest of the proof that we want to eliminate a later modality, not just the present step. For example, we can save a credit £1 from one step, keep it for two subsequent steps, and then use it before the next step to eliminate a later guarding an invariant assertion. In particular, the credit £1 can also be used as part of purely logical reasoning where there is no program in sight.

The reader may wonder whether later credits are really a backdoor for reintroducing into the logic the kind of explicit step-index manipulation that the later modality was designed to avoid. The answer is no: because unlike step-indices, later credits are implemented as *resources* in a separation logic, and hence they inherit all the modular reasoning principles associated with resources in a separation logic. To wit: if we "own" £1 (*i.e.,* it is in our precondition), then we alone get to decide how we want to spend it without any interference from other parts of the program/proof. If we want to spend it to eliminate a later, then we can do so with a credit spending rule. If we want to share it with other functions, then we can pass it to them as part of their precondition. If we want to keep it to ourselves during a function call, then we can frame it around the function call. If we want to share it with other threads, then we can put it into an invariant that is shared with those threads. In short, we can reason about later credits using all the standard reasoning patterns that are available for resources in separation logic. None of this is possible with step-index manipulation.

[1] We use nested invariants here, because they are one of the simpler examples to illustrate where the existing practices are not enough. In practice, most proofs do not need to use nested invariants. However, plenty of proofs put other (more complicated) step-indexed assertions into invariants, and then guarding laters cause trouble (*e.g.,* see §3 and §4).

The reasoning that later credits enable has two kinds of applications: First, it can simplify existing proofs. Later credits can help where step-indexing previously got in the way and cluttered the proofs. Second, the reasoning can enable proofs which were seemingly not possible with standard later elimination techniques. We will see examples of both kinds of applications in the paper.

Contributions. Our main contributions are later credits and the amortized step-indexing technique that they enable. We develop both as an extension of the step-indexed separation logic Iris. We explain later credits (in §2), discuss their soundness (in §5), and show how they complement existing approaches to eliminate multiple laters per step (in §6). We demonstrate the use of later credits with two flagship examples (one of each kind):

(1) *New proofs.* One interesting application of step-indexed logical relations in prior work has been in proving that expressions in higher-order stateful languages can be *reordered* [Krogh-Jespersen et al. 2017; Timany et al. 2018]. However, due to trouble with the later modality, the step-indexed logical relations of prior work can only handle very restricted forms of reordering operations with shared higher-order state (*e.g.,* ones using shared, but immutable state). In §3, we show how later credits make it possible to prove much more sophisticated reorderings, in particular for JavaScript-inspired promises.

(2) *Proof simplification.* One of the original *raisons d'être* of Iris was proving *logical atomicity* for concurrent data structures. Sadly, step-indexing has always caused trouble for logical atomicity, sometimes ruling out natural and perfectly valid proof strategies and requiring "ugly" workarounds [Jung 2019]. In §4, we show how to avoid such workarounds by instead using later credits to implement simpler and more intuitive proofs of logical atomicity.

Besides these flagship examples, we develop several smaller case studies to demonstrate usefulness of later credits (in §6). We develop a form of *prepaid* invariants, which can be opened around physically atomic instructions *without* a guarding later, and we show that later credits can be used to prove the kind of "reverse refinements" introduced by Svendsen et al. [2016] without requiring the transfinite step-indexing model that Svendsen et al. needed. We have mechanized later credits [Spies et al. 2022] and all of the above examples in Coq using the Iris Proof Mode [Krebbers et al. 2017b, 2018]. Further technical details are included in the supplementary material.

2 LATER CREDITS IN A NUTSHELL

In this section, we explain the key ideas behind later credits in Iris (in §2.2). Before we do so, we first review the principles of Iris upon which later credits are built (in §2.1). Readers well-versed in Iris can skip subsection §2.1, and proceed directly to §2.2.

2.1 An Iris Primer

As Iris is a step-indexed separation logic, its reasoning principles (excerpt shown in Figure 1) rest on two pillars: separation logic and step-indexing.

Separation logic. Iris offers the standard connectives of separation logic: separating conjunction $P * Q$, the points-to assertion $\ell \mapsto v$, and Hoare triples $\{P\}\ e\ \{v.\ Q\}$. The Hoare triple $\{P\}\ e\ \{v.\ Q\}$ expresses that under precondition P, the expression e is safe (*i.e.,* cannot get stuck), and if it terminates with value v, then it satisfies the postcondition Q. The distinguishing feature of *separation* logic is that propositions not only assert facts about the state of the program, but also convey *ownership* of said state. That is, if an expression e has $\ell \mapsto 42$ in its precondition, then it not only knows that ℓ currently stores 42 in the heap, but also that no other program part can modify ℓ

Separation Logic:

$$\textsc{Frame}\ \frac{\{P\}\ e\ \{v.\,Q\}}{\{P * R\}\ e\ \{v.\,Q * R\}} \qquad \textsc{UpdReturn}\ \ P \vdash |\!\!\Rrightarrow P \qquad \textsc{UpdBind}\ \ (|\!\!\Rrightarrow P) * (P \mathrel{-\!\!*} |\!\!\Rrightarrow Q) \vdash |\!\!\Rrightarrow Q \qquad \textsc{UpdExec}\ \frac{\{P\}\ e\ \{v.\,Q\}}{\{|\!\!\Rrightarrow P\}\ e\ \{v.\,Q\}}$$

Step-Indexing:

$$\textsc{LaterIntro}\ \ P \vdash \triangleright P \qquad \textsc{LaterMono}\ \frac{P \vdash Q}{\triangleright P \vdash \triangleright Q} \qquad \textsc{Löb}\ \ (\triangleright P \Rightarrow P) \vdash P \qquad \textsc{PureStep}\ \frac{\{P\}\ e_2\ \{v.\,Q\} \qquad e_1 \rightarrow_{\text{pure}} e_2}{\{\triangleright P\}\ e_1\ \{v.\,Q\}}$$

$$\textsc{Timeless}\ \frac{\{P * Q\}\ e\ \{v.\,R\} \qquad \text{timeless}(Q)}{\{P * \triangleright Q\}\ e\ \{v.\,R\}} \qquad \textsc{LaterExists}\ \frac{X \text{ non-empty}}{\triangleright \exists x : X.\,\Phi(x) \vdash \exists x : X.\,\triangleright \Phi(x)} \qquad \textsc{LaterSep}\ \ \triangleright(P * Q) \vdash \triangleright P * \triangleright Q$$

$$\textsc{InvAlloc}\ \frac{\boxed{R}^{\mathcal{N}} \vdash \{P\}\ e\ \{w.\,Q\}}{\{P * \triangleright R\}\ e\ \{w.\,Q\}} \qquad \textsc{InvOpen}\ \frac{\{\triangleright R * P\}\ e\ \{v.\,\triangleright R * Q\}_{\mathcal{E}\setminus\mathcal{N}} \qquad \mathcal{N} \subseteq \mathcal{E} \qquad e \text{ phys. atomic}}{\boxed{R}^{\mathcal{N}} \vdash \{P\}\ e\ \{v.\,Q\}_{\mathcal{E}}}$$

Fig. 1. A selection of Iris's proof rules.

Implementation	**Specification**
mk_counter () $\triangleq$ **ref**(0)	$\{\text{True}\}$ mk_counter () $\{c.\ \text{counter}(c, 0)\}$
get$(c) \triangleq\ !c$	$\{\text{counter}(c, n)\}\ \text{get}(c)\ \{m \in \mathbb{N}.\ \text{counter}(c, m) \land m \geq n\}$
inc$(c) \triangleq$ **FAA**$(c, 1)$	$\{\text{counter}(c, n)\}\ \text{inc}(c)\ \{m \in \mathbb{N}.\ \text{counter}(c, m + 1) \land m \geq n\}$

Fig. 2. Implementation and specification of a concurrent monotone counter.

while it owns $\ell \mapsto 42$. For example, in separation logic it is trivial to prove:

$$\frac{\{\text{True}\}\ f\ ()\ \{v.\,\text{True}\}}{\{r \mapsto 0\}\ r \leftarrow 42; f\ (); !r\ \{v.\,v = 42 * r \mapsto 42\}}$$

In our precondition, we have ownership of $r \mapsto 0$. This enables us to store the value 42 in r, leaving us with ownership of $r \mapsto 42$. The subsequent call to f does not interfere with this ownership, meaning we retain it over the duration of the call and can still use it afterwards. Then, since we have ownership of $r \mapsto 42$, we can prove the read $!r$ results in 42. This form of ownership reasoning is made possible with the characteristic rule of separation logic, the framing rule Frame, which we can use to frame $r \mapsto 42$ around the call of $f\ ()$, and get it back after the execution of $f\ ()$.

On top of the above ownership reasoning over heap fragments, Iris offers an additional form of ownership reasoning: reasoning about *resources*. Resources are a form of *ghost state*, state that is not physically present in the program but useful for its verification. Resources will be a vital ingredient in the later credits mechanism. To understand how they work, we consider an example: a fine-grained concurrent (*i.e.,* that does not use locking), monotone counter (*i.e.,* that only increases in value). As depicted in Figure 2, our counter offers three methods: we can create a counter with mk_counter (implemented as a reference internally), we can read its value with get (implemented by reading the reference), and we can increment its value with inc (implemented with the concurrency primitive **FAA**, which does an atomic fetch-and-add and returns the old value).

In Iris, we can specify this counter with a predicate $\mathsf{counter}(c, n)$, which expresses that the value of counter c is currently at least n. Importantly, $\mathsf{counter}(c, n)$ only expresses that the counter value is "at least n" and not "exactly n", because we are considering a *concurrent* counter. That is, the counter can be shared between threads, and, after we have observed the counter value (*e.g.,* with a get), other threads can increment it, invalidating any assumptions about its *exact* value (but not about lower bounds). We define the counter predicate with an invariant and resources:

$$\mathsf{counter}(c, n) \triangleq \exists \gamma.\ \boxed{\exists m : \mathbb{N}.\ c \mapsto m * \mathsf{mono}_\gamma(m)}^{\mathcal{N}} * \mathsf{lb}_\gamma(n)$$

We use the invariant $\boxed{\exists m : \mathbb{N}.\ c \mapsto m * \mathsf{mono}_\gamma(m)}^{\mathcal{N}}$ to share ownership of $c \mapsto m$ between threads. Each thread that knows about the invariant can open it for an atomic step to get ownership of $c \mapsto m$ and $\mathsf{mono}_\gamma(m)$, and has to return both after the step. Since the invariant assertion only represents *knowledge*, not exclusive ownership, it can be duplicated and, hence, shared with other threads. We already know the assertion $c \mapsto m$ inside the invariant conveys ownership of the location c, in this case storing the counter value m. The assertion $\mathsf{mono}_\gamma(m)$ is an exclusively owned resource with name γ. More precisely, it is a monotonically growing resource (*i.e., m* can only be increased), which ensures that the value stored at location c is never decreased. The duplicable assertion $\mathsf{lb}_\gamma(n)$, part of the definition of $\mathsf{counter}(c, n)$, is another resource. It is a lower bound on the monotonically growing resource $\mathsf{mono}_\gamma(m)$. Thus, if we own $\mathsf{mono}_\gamma(m)$ and $\mathsf{lb}_\gamma(n)$, then we can deduce $m \geq n$ (*i.e.,* $\mathsf{mono}_\gamma(m) * \mathsf{lb}_\gamma(n) \vdash m \geq n$). For our counter, this rule is the key to proving the specification of get, because it allows us to deduce that the return value is at least n.

Above, we have seen how to use ghost state in the form of resources to augment the physical state in our proofs. What we have not discussed yet is how we *modify* ghost state. For example, in the verification of inc, we need to update $\mathsf{mono}_\gamma(n)$ to $\mathsf{mono}_\gamma(n+1)$ to match the value of the counter. To modify resources, Iris has a designated modality: the update modality $\Rrightarrow P$. Intuitively, $\Rrightarrow P$ means P holds after (possibly) performing some updates to the resources.

Understanding the basics of the update modality is essential for understanding later credits, so we take a closer look. First, each resource type (like the monotone counter) comes with resource-specific update rules. For example, the update rule for monotone counters is $\mathsf{mono}_\gamma(n) \vdash \Rrightarrow \mathsf{mono}_\gamma(n+1) * \mathsf{lb}_\gamma(n+1)$—*i.e.,* we can update the ownership of $\mathsf{mono}_\gamma(n)$ to ownership of $\mathsf{mono}_\gamma(n+1)$ and $\mathsf{lb}_\gamma(n+1)$.

Besides the resource-specific rules, there are three important update rules: UpdReturn, UpdBind, and UpdExec. The first two rules essentially express that the update modality is a monad: we can construct a no-op update with UpdReturn (the monadic return), and we can compose two updates with UpdBind (the monadic bind). In the rule UpdBind, the so-called magic wand $P \mathrel{-\!\!*} Q$ may be understood as an implication which transfers ownership. In other words, if we own $P \mathrel{-\!\!*} Q$ and are willing to give up ownership of P, then we get back ownership of Q. The third rule, UpdExec, shows how we can *execute* the update modality (read bottom to top): if we have an update in our precondition (*e.g.,* from updating $\mathsf{mono}_\gamma(n)$ to $\mathsf{mono}_\gamma(n+1)$), then we can execute the update and proceed with reasoning about the precondition P. Put differently, whenever we are verifying programs with Hoare triples, we can update resources at will.

Step-indexing. The second pillar of Iris is step-indexing. Recall (from §1) that instead of reasoning explicitly about predicates over natural numbers, Iris takes the "logical approach" to step-indexing with the later modality $\triangleright P$ [Appel et al. 2007]. Intuitively, the later modality expresses that P will hold after the next program step. To illustrate how that works in practice, we consider an example: verifying partial correctness of an infinite loop in Iris. That is, we show $\Phi_{\mathrm{loop}} \triangleq \{\mathsf{True}\}\ \mathsf{loop}\ ()\ \{v.\ \mathsf{False}\}$, where $\mathsf{loop} \triangleq \mathbf{rec}\ \mathsf{loop}\ x = \mathsf{loop}\ x$.

To prove the triple Φ_{loop}, we need recursive reasoning. Unfortunately, because loop does not terminate, there is no inductive argument that we can use to prove Φ_{loop}. For such cases, step-indexed separation logics offer *Löb induction*, which can be understood as a coinduction principle. The Löb rule says that if we want to prove a property P, then we can assume the property P holds later (*i.e.,* $\triangleright P$). In this rule, the later modality ($\triangleright$) acts as a guard—it prevents us from using P directly (which would make proving any proposition trivial). To eliminate the guarding later, we can execute a program step. For example, we can apply the rule PureStep, which enables us to eliminate a later from the precondition when we take a pure step (*i.e.,* a step without non-determinism and state).

For our proof of the triple Φ_{loop}, Löb induction and the rule PureStep suffice. First, with Löb induction, we assume the triple Φ_{loop} already holds later (*i.e.,* we assume $\triangleright \Phi_{\mathsf{loop}}$) and continue to show Φ_{loop}. Next, we execute $\mathsf{loop}\ ()$ for one pure step using PureStep (since $\mathsf{loop}\ () \to_{\mathsf{pure}} \mathsf{loop}\ ()$). Afterwards, we are again left with the goal Φ_{loop}, but we have eliminated the guarding later from our assumption by taking a step. Thus, all that remains to prove is the trivial goal $\Phi_{\mathsf{loop}} \vdash \Phi_{\mathsf{loop}}$.

However, there is more to step-indexing and the later modality than just coinduction. Other parts of Iris can hook into Iris's step-indexing mechanism to resolve their own cyclic dependencies. In the introduction, we have already encountered one example: *impredicative invariants* [Svendsen and Birkedal 2014]. Recall that when we open an impredicative invariant[2] $\boxed{R}^{\mathcal{N}}$, then its contents R are guarded by a later (see InvOpen). The reason that later modalities show up here is that impredicative invariants are cyclic, and step-indexing is used to stratify these cycles in the model of Iris.

Once we have opened an invariant, we are confronted with a guarding later modality. Unfortunately, while introducing and reasoning under a later modality is easy (see LaterIntro and LaterMono), eliminating them can be challenging. Sometimes, we get lucky and the later eliminations align with program steps (*e.g.,* in the proof of Φ_{loop}). However, often, they do not. Then we still have two options available: *later commuting rules* and *timeless propositions*. We explain both with the counter from Figure 2. In the verification of get, we have to open the counter invariant and obtain $\triangleright(\exists n : \mathbb{N}.\ \ell \mapsto n * \mathsf{mono}_\gamma(n))$. Using the commuting rules LaterExists and LaterSep, we can move the later inwards and obtain $(\exists n : \mathbb{N}.\ \triangleright \ell \mapsto n * \triangleright \mathsf{mono}_\gamma(n))$. Since $\ell \mapsto n$ is timeless, meaning it is independent of Iris's step-indexing mechanism, we can use the rule Timeless and the fact $\mathsf{timeless}(\ell \mapsto v)$ to eliminate the guarding later from $\ell \mapsto n$. Without the guarding later, we can then use $\ell \mapsto n$ to justify the load from ℓ.

In some cases, however, none of these techniques (*i.e.,* eliminating laters after steps, commuting rules, and timelessness) apply. For such cases, we now introduce a fourth option: *later credits*.

2.2 Later Credits in Iris

The later credits mechanism (whose rules are shown in Figure 3) rests on two central pieces: a new resource $\pounds n$, called the *later credits*, and a new update modality $\Rrightarrow_{\mathsf{le}} P$, called the *later elimination update*. Intuitively, one can think of owning $\pounds n$ as the right to eliminate n later modalities, and of the later elimination update $\Rrightarrow_{\mathsf{le}} P$ as an extension of Iris's update modality that additionally allows updating $\triangleright P$ to P using later credits. The later credits mechanism factors into two parts:

(1) We *receive* later credits by taking program steps. For example, we receive one later credit $\pounds 1$ by executing a pure step with PureStep. After the program step, the new credit becomes available in the precondition of the Hoare triple of the successor expression e_2. (The proof

[2] As mentioned in §1, invariants carry a namespace $\mathcal{N}$ and Hoare triples carry a mask $\mathcal{E}$ (as do updates "$\Rrightarrow$", which we will explain in §3). This allows us to prevent reentrancy, *i.e.,* opening the same invariant twice at the same time—taking out the same resources from an invariant twice would lead to unsoundness. Namespaces and masks are orthogonal to later credits and clutter the presentation, so we omit them here and state the full rules in the supplementary material [Spies et al. 2022].

CreditSplit
$$\pounds(n+m) \Leftrightarrow \pounds n * \pounds m$$

CreditTimeless
$$\mathsf{timeless}(\pounds n)$$

PureStep
$$\frac{\{P * \pounds 1\}\; e_2\; \{v.\, Q\} \qquad e_1 \to_{\mathrm{pure}} e_2}{\{P\}\; e_1\; \{v.\, Q\}}$$

LEUpdLater
$$\pounds 1 * \triangleright P \vdash \mathrel{|}\!\!\Rrightarrow_{\mathsf{le}} P$$

LEUpdReturn
$$P \vdash \mathrel{|}\!\!\Rrightarrow_{\mathsf{le}} P$$

LEUpdBind
$$(\mathrel{|}\!\!\Rrightarrow_{\mathsf{le}} P) * (P \mathrel{-\!\!*} \mathrel{|}\!\!\Rrightarrow_{\mathsf{le}} Q) \vdash \mathrel{|}\!\!\Rrightarrow_{\mathsf{le}} Q$$

LEUpdExec
$$\frac{\{P\}\; e\; \{v.\, Q\}}{\left\{\mathrel{|}\!\!\Rrightarrow_{\mathsf{le}} P\right\}\; e\; \{v.\, Q\}}$$

Fig. 3. A selection of proof rules for later credits.

rules for load, store, allocation, *etc.* similarly generate one credit after the step.) Once we have received credits, we can combine and split them freely with CreditSplit.

(2) We *spend* later credits through later elimination updates $\mathrel{|}\!\!\Rrightarrow_{\mathsf{le}}$. That is, with LEUpdLater we can give up a credit $\pounds 1$, and, in exchange, eliminate a later modality by updating $\triangleright P$ to P. In particular, we can use this rule to eliminate a guarding later from one of our assumptions. Once we own $\mathrel{|}\!\!\Rrightarrow_{\mathsf{le}} P$, these updates can be executed as usual. For example, just like standard updates $\mathrel{|}\!\!\Rrightarrow P$, we can execute them in the precondition of Hoare triples with LEUpdExec.

Have we just managed to replace one modality ($\triangleright$) with another one ($\mathrel{|}\!\!\Rrightarrow_{\mathsf{le}}$)? No, far from it. There are two key distinctions between the two modalities. The first one is that $\mathrel{|}\!\!\Rrightarrow_{\mathsf{le}} P$ can be executed virtually everywhere in the logic, whereas the elimination of laters is quite restricted (as we have explained above). To integrate $\mathrel{|}\!\!\Rrightarrow_{\mathsf{le}} P$ into Iris, we replace the update $\mathrel{|}\!\!\Rrightarrow P$ with $\mathrel{|}\!\!\Rrightarrow_{\mathsf{le}} P$ in most of Iris. This modification allows us to execute $\mathrel{|}\!\!\Rrightarrow_{\mathsf{le}} P$ everywhere that we could execute $\mathrel{|}\!\!\Rrightarrow P$ before.

The second key distinction is that ($\mathrel{|}\!\!\Rrightarrow_{\mathsf{le}}$) is more compositional. Analogous to ($\mathrel{|}\!\!\Rrightarrow$), ($\mathrel{|}\!\!\Rrightarrow_{\mathsf{le}}$) is a monad with LEUpdReturn and LEUpdBind, whereas ($\triangleright$) is not (it is only an applicative functor). Since it is a monad, we can use LEUpdBind to accumulate and compose updates. For example, it is trivial to prove transitivity (*i.e.,* $\mathrel{|}\!\!\Rrightarrow_{\mathsf{le}} \mathrel{|}\!\!\Rrightarrow_{\mathsf{le}} P \vdash \mathrel{|}\!\!\Rrightarrow_{\mathsf{le}} P$), or to use one later elimination update to spend *two* credits and eliminate *two* laters. In contrast, ($\triangleright$) does not satisfy the analogous rule $\triangleright\triangleright P \vdash \triangleright P$, so we cannot fold two laters into one.

Later credits in action. As a first illustration of later credits, we show how to save a credit for a few steps to enable a later elimination afterwards. We do not *need* later credits here, because we could use timelessness and later commuting for this example (see §2.1), but it will nevertheless be instructive as a toy example:

$$\frac{\forall n.\, \{n \in \mathbb{N}\}\; f\, n\; \{m.\, m \in \mathbb{N}\}}{\boxed{\exists n : \mathbb{N}.\, \ell \mapsto n}^{\mathcal{N}} \vdash \{\mathsf{True}\}\; l \leftarrow f(41+1)\; \{v.\, \mathsf{True}\}}$$

We execute $41 + 1$ with PureStep and thereby obtain a new later credit $\pounds 1$. We are left with proving $\boxed{\exists n : \mathbb{N}.\, \ell \mapsto n}^{\mathcal{N}} \vdash \{\pounds 1\}\; l \leftarrow f(42)\; \{v.\, \mathsf{True}\}$. We frame $\pounds 1$ around the call of f with Frame, leaving us to prove $\boxed{\exists n : \mathbb{N}.\, \ell \mapsto n}^{\mathcal{N}} \vdash \{\pounds 1\}\; l \leftarrow m\; \{v.\, \mathsf{True}\}$ for some $m \in \mathbb{N}$. After opening the invariant with InvOpen, we have to show $\{\pounds 1 * \triangleright(\exists n : \mathbb{N}.\, \ell \mapsto n)\}\; l \leftarrow m\; \{v.\, \triangleright(\exists n : \mathbb{N}.\, \ell \mapsto n)\}$. We spend the later credit to eliminate the later modality with LEUpdLater, leaving us to prove $\left\{\mathrel{|}\!\!\Rrightarrow_{\mathsf{le}} (\exists n : \mathbb{N}.\, \ell \mapsto n)\right\}\; l \leftarrow m\; \{v.\, \triangleright(\exists n : \mathbb{N}.\, \ell \mapsto n)\}$. Subsequently, we execute the later elimination update with LEUpdExec, leaving us to prove $\{\exists n : \mathbb{N}.\, \ell \mapsto n\}\; l \leftarrow m\; \{v.\, \triangleright(\exists n : \mathbb{N}.\, \ell \mapsto n)\}$. The rest of the proof is routine, using LaterIntro in the postcondition.

Although it is simple, this example shows how later credits enable reasoning about later eliminations as an ownable resource, which can be passed around using the rules of separation logic. This

$$\begin{aligned}
\texttt{promise} &: 1 \rightarrow \mathsf{pr}(\tau) \\
\texttt{promise}() &\triangleq (\texttt{mklock}(), \textbf{ref}(\mathsf{none}), \textbf{ref}([])) \\
\texttt{resolve} &: \mathsf{pr}(\tau) \times \tau \rightarrow 1 \\
\texttt{resolve}((l, r, c), a) &\triangleq \texttt{lock}(l); \textbf{case}\ !r\ \textbf{of}\ \mathsf{some}(b) \Rightarrow \texttt{unlock}(l); \textbf{abort}() \\
&\qquad\qquad\qquad \mid \mathsf{none} \Rightarrow r \leftarrow \mathsf{some}(a); \textbf{let}\ fs = !c;\ c \leftarrow [\,]; \\
&\qquad\qquad\qquad\qquad\quad \texttt{unlock}(l);\ \mathsf{app}\ (\lambda f.\, f(a))\ fs \\
\texttt{then} &: \mathsf{pr}(\tau) \times (\tau \rightarrow 1) \rightarrow 1 \\
\texttt{then}((l, r, c), f) &\triangleq \texttt{lock}(l); \textbf{case}\ !r\ \textbf{of}\ \mathsf{some}(a) \Rightarrow \texttt{unlock}(l); f(a) \\
&\qquad\qquad\qquad \mid \mathsf{none} \Rightarrow c \leftarrow f :: !c; \texttt{unlock}(l)
\end{aligned}$$

Fig. 4. Promise implementation in HeapLang.

kind of reasoning is essential in the examples that follow, in which we frame credits for several steps (in §3) and even exchange them through invariants (in §4 and §6.1).

3 LATER CREDITS FOR REORDERING REFINEMENTS

For our first application of later credits, we show how they address a limitation of step-indexed logical relations that arises when proving *reordering refinements*. In a reordering refinement, we want to prove that two expressions e_1 and e_2 are independent in the sense that their execution order is not observable. Concretely, this means we want to show:

$$e_2; e_1 \leq_{\text{ctx}} e_1; e_2 \qquad \textit{and, more generally,} \qquad e_1 \parallel e_2 \leq_{\text{ctx}} (e_1, e_2)$$

where $e_1 \parallel e_2$ denotes the parallel composition of e_1 and e_2 (returning the pair of their result values). One way to prove such a contextual refinement is with a step-indexed logical relation. That is, for $e_1, e_2 : \tau$ we show $e_1 \parallel e_2 \leq_{\log} (e_1, e_2) : \tau \times \tau$ where $\leq_{\log}$ is a step-indexed relation implying $\leq_{\text{ctx}}$.

However, with step-indexed logical relations, it is difficult to prove reordering refinements involving shared higher-order state. This is unfortunate, since such state is one of the main reasons for using step-indexing in the first place. The difficulty arises because laters are eliminated in these relations *asymetrically*: when proving a logical refinement of the form $e \leq_{\log} e' : \tau$, elimination of laters is only allowed during steps on the left (steps of e). However, for a reordering refinement like $e_2; e_1 \leq_{\log} e_1; e_2 : 1$, the laters eliminated when stepping e_2 on the left could be too "early" to help with eliminating laters needed for reasoning about e_2 on the right. Later credits resolve this issue by letting us save credits from the execution of e_2 on the left and use them when reasoning about e_2 on the right. This enables us to reorder operations that use shared, mutable, higher-order state, which is beyond the scope of previous work [Krogh-Jespersen et al. 2017; Timany et al. 2018].

A motivating example. We will develop a general proof technique for reordering refinements, but to stay concrete, let us focus here on a specific example where later credits will be essential for the proof: *promises*. A promise, in languages such as JavaScript, represents the result of a delayed computation—the value is not available right away, but it is "promised" to be there eventually.

We can implement a simplified version of the mechanism (depicted in Figure 4) in Iris's HeapLang. We promise a value of type τ with `promise`, select the value for the promise with `resolve`, and attach continuations $f : \tau \rightarrow 1$ to a promise with `then`, which will be executed once the promise has been resolved. Internally, a promise consists of a reference for the result r, a list of continuations c, and a lock l to protect the two. When a promise is resolved, the value is stored in r and all continuations in c are executed with the function app. (We explain the **abort** case shortly.) The operation then adds the continuation to the list if the promise is unresolved, or executes it with the value of the promise.

What is interesting about promises is that (under suitable conditions) their operations can be reordered. For example, if two continuations f and g are reorderable with respect to each other, then the order in which we attach them to a promise using then does not really matter. Similarly, if a promise is only ever resolved once, the order in which we call then and resolve does not matter either, since the attached callback will eventually be executed with the resolved value of the promise. Thus, for reorderable f and g, we should be able to prove, for example:

$$\texttt{then}(p, f); \texttt{then}(p, g); \texttt{resolve}(p, a) \leq_{\text{ctx}} \texttt{then}(p, g); \texttt{resolve}(p, a); \texttt{then}(p, f) : 1$$

To state this refinement precisely, we need to formalize the conditions on the use of then and resolve. To model when two functions f and g are reorderable, we will use a type system (in §3.2). And as far as resolve is concerned, resolving a promise twice is typically considered an error. For example, in JavaScript a repeated resolve attempt has no effect. Thus, we simply rule out multiple resolve attempts in the promise implementation: we implement a second call to resolve as "*safe-failure*" (via **abort**, which just diverges).

But even once we have formalized these constraints, proving such a refinement remains challenging because the continuations stored in memory are a form of shared, higher-order state. That is where later credits come into the picture. They will enable us to construct a logical relation that can nevertheless prove this refinement. To do so, we start with Iris's standard binary logical relation, ReLoC [Frumin et al. 2018, 2021] (in §3.1). We extend ReLoC with support for proving reorderings by adapting ideas from Timany et al. [2018] (in §3.2). Finally, we show that later credits allow us to prove promise reorderings (in §3.3).

3.1 ReLoC: Logical Relations in Iris

ReLoC uses Iris's program logic to define a logical relation. This requires a way to do relational reasoning about pairs of programs in Iris, instead of the unary reasoning about a single program that we have seen so far. To do relational reasoning, ReLoC uses a technique from CaReSL [Turon et al. 2013], in which the "specification" program (on the right-hand side of the refinement) is represented by ghost state—as a *ghost program* so to speak (this technique has also been used in other binary logical relations in Iris [Krogh-Jespersen et al. 2017; Krebbers et al. 2017b; Tassarotti et al. 2017; Timany et al. 2018; Spies et al. 2021]). The ghost program has ghost state assertions of the form $j \Mapsto e$, which mean that thread j in the ghost program is executing expression e, and assertions of the form $\ell \mapsto_{\mathrm{s}} v$, which mean that location ℓ points to v in the ghost program's memory (the subscript s here stands for "specification"). The ghost program is executed by updating the ghost assertions with the update modality. For example, to perform a store of w to location ℓ in the ghost program, we have the rule $j \Mapsto (\ell \leftarrow w) * \ell \mapsto_{\mathrm{s}} v \vdash \Rrightarrow^{\mathcal{N}_{\text{reloc}}} j \Mapsto () * \ell \mapsto_{\mathrm{s}} w$, which reflects that the store returns $()$ and the location stores the value w afterwards.

One thing we have not seen so far is a mask (here $\mathcal{N}_{\text{reloc}}$) on an update "$\Rrightarrow$".[3] In Iris, updates with masks enable invariants to be opened as part of ghost state reasoning (*e.g.*, in proving the update to the contents of ghost location ℓ above) rather than just around steps of computation:

$$\textsc{InvOpenUpd}\quad \frac{P * \triangleright R \vdash \Rrightarrow^{\mathcal{E} \setminus \mathcal{N}} Q * \triangleright R \qquad \mathcal{N} \subseteq \mathcal{E}}{P * \boxed{R}^{\mathcal{N}} \vdash \Rrightarrow^{\mathcal{E}} Q} \qquad\qquad \textsc{UpdMaskWeaken}\quad \frac{\mathcal{E}_1 \subseteq \mathcal{E}_2 \qquad P \vdash \Rrightarrow^{\mathcal{E}_1} P}{P \vdash \Rrightarrow^{\mathcal{E}_2} P}$$

[3] In other Iris presentations, the mask for a non-mask-changing update is in subscript (*i.e.*, "$\Rrightarrow_{\mathcal{E}}$"). Since this position conflicts with the "le" of our later elimination updates, we change it to superscript.

$$
\begin{aligned}
[\![\texttt{int}]\!] &\triangleq \lambda(v_1, v_2).\ \exists z \in \mathbb{Z}.\ v_1 = v_2 = z \\
[\![\tau \to \tau']\!] &\triangleq \lambda(v_1, v_2).\ \forall u_1, u_2.\ \Box([\![\tau]\!](u_1, u_2) \mathrel{-\!\!*} (v_1\ u_1) \le (v_2\ u_2) : [\![\tau']\!]) \\
[\![\mathsf{ref}\ \tau]\!] &\triangleq \lambda(v_1, v_2).\ \exists \ell_1, \ell_2.\ v_1 = \ell_1 * v_2 = \ell_2 * \boxed{\exists u_1, u_2.\ \ell_1 \mapsto u_1 * \ell_2 \mapsto_{\mathsf{s}} u_2 * [\![\tau]\!](u_1, u_2)}^{\mathcal{N}.\ell_1.\ell_2}
\end{aligned}
$$

Open Expressions

Let $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$.

$$(\Gamma \vDash e \le_{\log} e' : \tau) \triangleq \forall v_1, v_1', \ldots, v_n, v_n'.$$

$$\left(\ast_{i=1,\ldots,n}\ [\![\tau_i]\!](v_i, v_i') \right) \vdash e[v_1/x_1] \cdots [v_n/x_n] \le_{\log} e'[v_1'/x_1] \cdots [v_n'/x_n] : [\![\tau]\!]$$

Fig. 5. Excerpt of ReLoC.

In the ghost program rule, the namespace $\mathcal{N}_{\text{reloc}}$ is only an implementation detail of ReLoC. Nevertheless, we emphasize the ability to open invariants around updates here, because it will be crucial in our example refinements (in §3.3).

In ReLoC, to prove a relational property about programs e_1 and e_2, it suffices to prove a Hoare triple for e_1 in which the precondition has a ghost thread running e_2 in an arbitrary evaluation context K. ReLoC defines a binary relation in Iris that expresses this pattern for an arbitrary postcondition $P : (\mathit{Val} \times \mathit{Val}) \to \mathit{iProp}$, where iProp is the type of Iris assertions:

$$(e_1 \le e_2 : P) \triangleq \forall j, K.\ \{j \Rrightarrow K[e_2]\}\ e_1\ \{v_1.\ \exists v_2.\ j \Rrightarrow K[v_2] * P(v_1, v_2)\}$$

The adequacy theorem of Iris then ensures that, if $\mathsf{True} \vdash e_1 \le e_2 : P$ and e_1 terminates with value v_1, there exists an execution of e_2 in which it terminates with a value v_2 such that $P(v_1, v_2)$ holds.

This definition of $(\le)$ treats the programs e_1 and e_2 asymmetrically. In particular, since the Hoare triple is about e_1, steps of e_1 get to eliminate laters. In contrast, e_2 is just a ghost program, so as it is executed, no laters are eliminated. Typically, this asymmetry is not a problem, because we are reasoning about the two programs "in sync": by taking steps of e_1 at the same time as we perform steps of e_2, we can use the physical steps of e_1 to eliminate laters needed for reasoning about e_2.

Given the binary relation $e_1 \le e_2 : P$, defining the logical relation $\Gamma \vDash e_1 \le_{\log} e_2 : \tau$ is relatively straightforward (depicted in Figure 5). First, we define a type interpretation $[\![-]\!] : \mathit{Type} \to (\mathit{Val} \times \mathit{Val}) \to \mathit{iProp}$, which maps every type τ to an Iris relation on values $[\![\tau]\!]$. Then, we define $(\le_{\log})$ by lifting $(\le)$ to open expressions. To simplify the explanation here, we leave out the details of how this approach scales to polymorphic and recursive types.

In the definition of $[\![-]\!]$, the interesting cases are $\tau \to \tau'$ and $\mathsf{ref}\ \tau$. The former says that two values are related at type $\tau \to \tau'$ if, whenever they are applied to values related at type τ, the resulting application expressions are related at the interpretation of τ'. In this case, we use Iris's *persistence* modality "$\Box$" to ensure values of type $\tau \to \tau'$ can be used multiple times. In general, $\Box P$ makes sure that P holds and that its proof does not depend on any exclusive resources.

For $\mathsf{ref}\ \tau$, the relation says that the two values must be locations, and we use an Iris invariant assertion that requires the two locations to always point to values that are related at type τ. The invariant here is implicitly making use of Iris's step-indexing, which is what allows us to avoid the usual circularity issues that arise in defining logical relations for systems with higher-order mutable state [Ahmed 2004; Birkedal et al. 2011].

The logical relation has the following two key properties:

Theorem 3.1 (Soundness). *If* $\Gamma \vDash e_1 \le_{\log} e_2 : \tau$, *then* $\Gamma \vdash e_1 \le_{\text{ctx}} e_2 : \tau$.

Theorem 3.2 (Fundamental Property). *If* $\Gamma \vdash e : \tau$*, then* $\Gamma \vDash e \leq_{\mathsf{log}} e : \tau$*.*

The soundness theorem is what ensures that the logical relation is useful for proving contextual equivalences, and it follows from the adequacy of Iris. Meanwhile, the fundamental property lets us automatically deduce that a syntactically well-typed term is logically related to itself.[4] This theorem is proven by showing that the logical relation is a congruence w.r.t. all typing rules.

3.2 Reorderability Extension

Next, we add reorderability. The type system of ReLoC is not rich enough to state that a function is reorderable. To reason about reorderings, we extend the type system with a new type $\tau_1 \rightarrow_{\mathsf{re}} \tau_2$ of reorderable functions. The typing rule for this type uses a new relation, $\Gamma \vdash^{\mathsf{re}} e : \tau$, which implies that e is a *reorderable* expression of type τ. This judgment is a restriction of the standard typing judgment $\vdash$ that removes the rules for operations that have side effects. We then extend the standard typing judgment $\vdash$ with two new rules for introducing and eliminating terms of type $\rightarrow_{\mathsf{re}}$:

$$\frac{\Gamma, x : \tau_1, f : \tau_1 \rightarrow_{\mathsf{re}} \tau_2 \vdash^{\mathsf{re}} e : \tau_2}{\Gamma \vdash (\mathbf{rec}\, f\, x = e) : \tau_1 \rightarrow_{\mathsf{re}} \tau_2} \qquad \frac{\Gamma \vdash f : \tau_1 \rightarrow_{\mathsf{re}} \tau_2 \qquad \Gamma \vdash e : \tau_1}{\Gamma \vdash f\ e : \tau_2}$$

The fragment $\vdash^{\mathsf{re}}$ is fairly limited, since expressions may not contain instructions with side effects. However, it is possible to bend this limitation. In the following, we will develop a logical relation for $\vdash^{\mathsf{re}}$ which admits additional terms that cannot be typed syntactically in the side-effect free fragment $\vdash^{\mathsf{re}}$, but are *semantically reorderable*. For example, $\lambda\_.\ \mathbf{let}\ r = \mathbf{ref}(41); !r + 1$ and `then` are semantically reorderable even though they have side effects. To extend the logical relation to support the reorderability type judgment ($\vdash^{\mathsf{re}}$) and reorderable function type ($\rightarrow_{\mathsf{re}}$), we take inspiration from Timany et al. [2018] to define a reorderable form of $e_1 \leq e_2 : P$ as follows:

$$(e_1 \leq^{\mathsf{re}} e_2 : P) \triangleq \{\mathsf{True}\}\ e_1\ \left\{v_1.\ \exists v_2.\ P(v_1, v_2) * e_2 \rightsquigarrow_{\mathsf{ghost}} v_2\right\}$$
$$\textit{where}\ \ e_1 \rightsquigarrow_{\mathsf{ghost}} e_2 \triangleq \forall j, K.\ j \Rrightarrow K[e_1] \mathrel{-\!\!*} \mathrel{|\!\!\Rrightarrow}^{\top}_{\mathsf{le}}\ j \Rrightarrow K[e_2]$$

The key difference between $\leq$ and $\leq^{\mathsf{re}}$ is that in the latter, the execution of the ghost program is moved entirely to the postcondition, as captured by the $\rightsquigarrow_{\mathsf{ghost}}$ assertion. That is, instead of executing the ghost program and the implementation "in sync" (as with the usual $\leq$ in ReLoC), we wait to run e_2 until *after* e_1 finishes running. This means e_1 executes "independently" of e_2, and subsequently e_2 executes independently of e_1. Disentangling e_1 and e_2 makes their executions reorderable, as we will see below. However, it also means physical steps of e_1 no longer directly eliminate laters that come up when reasoning about e_2. This could pose a problem if e_2 needs to take non-timeless resources out of an invariant. Fortunately, because ($\rightsquigarrow_{\mathsf{ghost}}$) uses a *later elimination update* "$\mathrel{|\!\!\Rrightarrow}^{\top}_{\mathsf{le}}$" instead of a standard update "$\mathrel{|\!\!\Rrightarrow}^{\top}$", we can spend later credits generated by e_1 as we execute ghost steps of e_2.

We integrate reorderability into ReLoC by defining

$$[\![\tau \rightarrow_{\mathsf{re}} \tau']\!] \triangleq \lambda v_1, v_2.\ \forall u_1, u_2.\ \Box([\![\tau]\!](u_1, u_2) \mathrel{-\!\!*} (v_1\ u_1) \leq^{\mathsf{re}} (v_2\ u_2) : [\![\tau']\!])$$

and lifting $e_1 \leq^{\mathsf{re}} e_2 : P$ to a version on open expressions $\Gamma \vDash e_1 \leq^{\mathsf{re}}_{\mathsf{log}} e_2 : \tau$ analogous to ($\leq_{\mathsf{log}}$). The new relation $\Gamma \vDash e_1 \leq^{\mathsf{re}}_{\mathsf{log}} e_2 : \tau$ ties neatly in with the standard ReLoC setup (from §3.1):

Lemma 3.3.

(1) *If* $\Gamma \vdash^{\mathsf{re}} e : \tau$*, then* $\Gamma \vDash e \leq^{\mathsf{re}}_{\mathsf{log}} e : \tau$*.*

(2) *If* $\Gamma \vDash e_1 \leq^{\mathsf{re}}_{\mathsf{log}} e_2 : \tau$*, then* $\Gamma \vDash e_1 \leq_{\mathsf{log}} e_2 : \tau$*.*

[4] As is common for binary logical relations, a term e being self-related means that it is "well-behaved". In this case, it means that e behaves like a syntactically well-typed term of type τ.

Moreover, reorderability ($\leq_{\mathrm{log}}^{\mathrm{re}}$) is strong enough to show the reorderings that we are after:

Lemma 3.4 (Reorderings).

(1) *If* $\Gamma \vDash e_1 \leq_{\mathrm{log}} e_1' : \tau_1$ *and* $\Gamma \vDash e_2 \leq_{\mathrm{log}}^{\mathrm{re}} e_2' : \tau_2$, *then* $\Gamma \vDash e_2 \parallel e_1 \leq_{\mathrm{log}} (e_2', e_1') : \tau_2 \times \tau_1$.
(2) *If* $\Gamma \vDash e_1 \leq_{\mathrm{log}} e_1' : 1$ *and* $\Gamma \vDash e_2 \leq_{\mathrm{log}}^{\mathrm{re}} e_2' : 1$, *then* $\Gamma \vDash e_2; e_1 \leq_{\mathrm{log}} e_1'; e_2' : 1$.

The proofs of both statements use the fact that e_2 can run *unconditionally* on the left (since the precondition of ($\leq_{\mathrm{log}}^{\mathrm{re}}$) is True) and that we can then delay execution of e_2' on the right arbitrarily. For example, consider the first statement. In the proof, we must verify a Hoare triple for the left term, $e_2 \parallel e_1$, which assumes ghost ownership of the right term in its precondition (*i.e.*, $j \Mapsto K[(e_2', e_1')]$). To verify $e_2 \parallel e_1$, we use the parallel composition rule of concurrent separation logic, which allows us to verify e_1 and e_2 separately, so long as we can split the precondition between them (*e.g.*, so long as we only give ownership of $j \Mapsto K[(e_2', e_1')]$ to one of them). In the case of e_1, the assumption $\Gamma \vDash e_1 \leq_{\mathrm{log}} e_1' : \tau_1$ requires ghost ownership of $j \Mapsto K'[e_1']$ (for some K'), so that e_1 can execute "in sync" with e_1'. Fortunately, we own $j \Mapsto K[(e_2', e_1')]$, so we can simply instantiate $K' := K[(e_2', \bullet)]$ and hand ownership of this assertion to the verification of e_1, which will produce $j \Mapsto K[(e_2', v_1')]$ (for some value v_1') in its postcondition.[5] Meanwhile, since e_2 is reorderable, its verification does not require any ghost code resource in its precondition, but rather produces the *delayed* ghost execution $e_2' \rightsquigarrow_{\mathrm{ghost}} v_2'$ (for some value v_2') in its postcondition. Finally, combining $e_2' \rightsquigarrow_{\mathrm{ghost}} v_2'$ and $j \Mapsto K[(e_2', v_1')]$, we can execute the delayed ghost execution and obtain $j \Mapsto K[(v_2', v_1')]$ as desired.

3.3 Promises with Later Credits

Equipped with the notion of reorderability, we return to our motivating example: *reordering promise operations.* Our main result for promises will be that their operations are in the logical relation:

Lemma 3.5 (Promise Typing).

(1) $\vDash \texttt{promise} \leq_{\mathrm{log}} \texttt{promise} : 1 \to \mathsf{pr}(\tau)$
(2) $\vDash \texttt{resolve} \leq_{\mathrm{log}}^{\mathrm{re}} \texttt{resolve} : \mathsf{pr}(\tau) \times \tau \to_{\mathrm{re}} 1$
(3) $\vDash \texttt{then} \leq_{\mathrm{log}}^{\mathrm{re}} \texttt{then} : \mathsf{pr}(\tau) \times (\tau \to_{\mathrm{re}} 1) \to_{\mathrm{re}} 1$

The proof of this lemma is *challenging* in terms of reasoning about different interleavings of the promise operations, but *simple* in terms of the later credits reasoning. Since we are mainly interested in later credits, we will give a detailed description of the use of later credits (in §3.3.3), and only a high-level description of the rest of the proof with examples (in §3.3.1 and §3.3.2). Let us start with an example of the kind of reorderings we can prove with this lemma and Lemma 3.4:

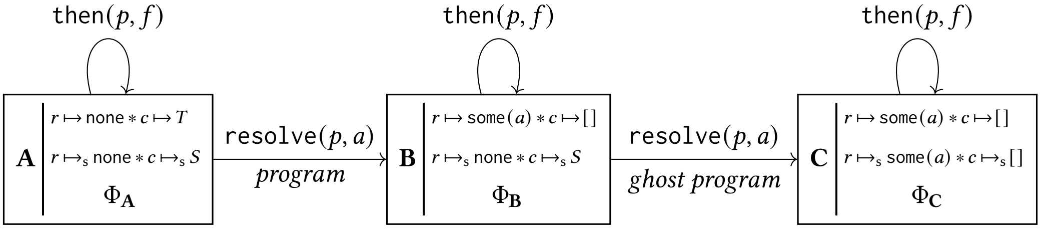
Corollary 3.6.

$$\frac{\Gamma \vDash e \leq_{\mathrm{log}} e : 1 \qquad p : \mathsf{pr}(\tau) \in \Gamma \qquad f : \tau \to_{\mathrm{re}} 1 \in \Gamma}{\Gamma \vDash \texttt{then}(p, f); e \leq_{\mathrm{log}} e; \texttt{then}(p, f) : 1}$$

There are two things to note about this corollary. First, when we prove that an expression like `then` is reorderable, we can move its execution *earlier.* This includes moving it across an arbitrary expression e, which could include additional calls to `then` or to `resolve`. Second, the corollary demonstrates the higher-order nature of the promise operations: `then` takes an arbitrary reorderable function f as its argument. In particular, f could resolve another promise q or (reentrantly) attach an additional continuation to p, since the operations `then` and `resolve` are reorderable.

[5] Note that we are using right-to-left evaluation order here, so e_1' executes before e_2'.

3.3.1 Promise Extension. Before we can prove Lemma 3.5, we have to extend the type interpretation $[\![-]\!]$ to include promises $\mathsf{pr}(\tau)$. The exact definition of $[\![\mathsf{pr}(\tau)]\!]$ is quite a mouthful and can be found in the supplementary material [Spies et al. 2022]. It consists of a lock and an invariant, which together encode a transition system for each promise. Here, we focus on the transition system:



Initially, in state **A**, the promise is unresolved in the program and the ghost program—the reference r is none, and the reference c is amassing continuations. The proposition Φ in each state stores some additional data that we need for the verification (*e.g.*, $\Phi_{\mathbf{A}}$ stores $[\![\tau \to_{\mathsf{re}} 1]\!](f, f)$ for each $f \in T$). We will return to Φ when we discuss a concrete example (in §3.3.2). From state **A**, we transition to state **B** when $\mathtt{resolve}(p, a)$ is executed in the program. We store a in the promise in the program but not yet in the ghost program, and we execute the continuations in T. From state **B**, we transition to state **C** when $\mathtt{resolve}(p, a)$ is executed in the ghost program. We store a in the reference and execute the continuations in S. If $\mathtt{then}(p, f)$ is executed in either the program or the ghost program, we do not change the state in the transition system. Depending on whether the promise has been resolved yet or not, f is either stored in the continuation list or directly executed.

3.3.2 The Continuation Exchange. Let us return to Lemma 3.5. We focus on the "continuation exchange", the key step in the proof where later credits will become necessary (in §3.3.3). To explain the continuation exchange, we use an instance of Corollary 3.6 as an example:

$$p : \mathsf{pr}(\tau), f : \tau \to_{\mathsf{re}} 1, a : \tau \vDash \mathtt{then}(p, f); \mathtt{resolve}(p, a) \leq_{\log} \mathtt{resolve}(p, a); \mathtt{then}(p, f) : 1$$

In this reordering, f is executed in different operations on both sides: on the left, $f(a)$ is executed in $\mathtt{resolve}(p, a)$ and on the right, in $\mathtt{then}(p, f)$. Proving this one specific reordering directly is not difficult. But Lemma 3.5 is challenging because to establish the semantic typing judgments, we must prove *local* Hoare triples about each individual operation without knowing the global set of operations on a given promise. To support this local reasoning, we introduce a scheme to "transfer" the ghost execution of $f(a)$ from `resolve` to `then`. This exchange is facilitated through the transition system. To describe the exchange, we sketch the cases of Lemma 3.5, but to simplify the explanation, we omit everything that is not relevant for the example. Remember that in each case, we first prove a Hoare triple about the program, and then subsequently show that the ghost program can be executed at any later point.

The resolve case. When $\mathtt{resolve}(p, a)$ is executed in the program (after $\mathtt{then}(p, f)$), we are in state **A** and we find the continuation $f \in T$. In this state, $\Phi_{\mathbf{A}}$ ensures $[\![\tau \to_{\mathsf{re}} 1]\!](f, f)$. We store a in r, transition to state **B**, and proceed to execute $f(a)$ in the program using $[\![\tau \to_{\mathsf{re}} 1]\!](f, f)$. Once we reach the end of $\mathtt{resolve}(p, a)$, we have executed $f(a)$ using $[\![\tau \to_{\mathsf{re}} 1]\!](f, f)$, and hence we own the ghost execution $f(a) \rightsquigarrow_{\text{ghost}} ()$. We use it to reason about $\mathtt{resolve}(p, a)$ in the ghost program. In this example (assuming S was initially empty), the ghost program observes that the continuation list S is empty (in state **B**), so there is nothing to execute. It stores a in r, thus advancing to state **C**. In this step, it stores the ghost execution $f(a) \rightsquigarrow_{\text{ghost}} ()$ in the transition system in $\Phi_{\mathbf{C}}$.

The then case. When $\mathtt{then}(p, f)$ is executed in the program, we are in state **A**. We store f in T, and we store $[\![\tau \to_{\mathsf{re}} 1]\!](f, f)$ in $\Phi_{\mathbf{A}}$ (see the resolve case). Afterwards, when $\mathtt{then}(p, f)$ is executed in the ghost program, we are in state **C**. The promise has already been resolved and $\mathtt{then}(p, f)$ will directly execute $f(a)$. Consequently, as part of proving $\mathtt{then}(p, f) \rightsquigarrow_{\mathrm{ghost}} ()$, we need to establish the ghost execution $f(a) \rightsquigarrow_{\mathrm{ghost}} ()$. Since $f(a) \rightsquigarrow_{\mathrm{ghost}} ()$ is stored in $\Phi_{\mathbf{C}}$ at this point, we want to use it to finish the execution. This *almost* works, but a "$\triangleright$" gets in the way ...

3.3.3 Using Later Credits. In the last step, we run into a "$\triangleright$" when we try to use the $f(a) \rightsquigarrow_{\mathrm{ghost}} ()$ stored in $\Phi_{\mathbf{C}}$. Recall that the transition system is encoded in an *invariant*, which we will denote $\boxed{\mathsf{TS}}^{\mathcal{N}}$ in the following. Hence, when we access the invariant in the final step, we only get $\triangleright\, f(a) \rightsquigarrow_{\mathrm{ghost}} ()$ out, but we need $f(a) \rightsquigarrow_{\mathrm{ghost}} ()$ to finish the proof, so (without later credits) we are stuck.

Let us zoom in on that last proof step. Formally, at that point in the proof, we have to show:

$$\boxed{\mathsf{TS}}^{\mathcal{N}} \vdash \mathtt{then}(p, f) \rightsquigarrow_{\mathrm{ghost}} ()$$

and since the promise is resolved in the ghost program at that point, we will be in state **C**. We may open $\boxed{\mathsf{TS}}^{\mathcal{N}}$ because of the update in $(\rightsquigarrow_{\mathrm{ghost}})$, but when we open $\boxed{\mathsf{TS}}^{\mathcal{N}}$, we will only get access to $\triangleright\, \mathsf{TS}$ (see InvOpenUpd), which effectively means we only get $\triangleright\, f(a) \rightsquigarrow_{\mathrm{ghost}} ()$. To prove $\mathtt{then}(p, f) \rightsquigarrow_{\mathrm{ghost}} ()$, we need to eliminate the later, but there are no *program* steps around to justify an elimination—only ghost program steps (which do not allow later elimination). We are stuck![6]

With later credits, the solution is simple. When we reason about $\mathtt{then}$ (*i.e.,* when we prove $\vDash \mathtt{then} \leq^{\mathsf{re}}_{\mathsf{log}} \mathtt{then} : \mathsf{pr}(\tau) \times (\tau \rightarrow_{\mathsf{re}} 1) \rightarrow_{\mathsf{re}} 1$), the execution *in the program* has plenty of steps that generate credits that we do not need—for example, the initial β-reduction step of $\mathtt{then}(p, f)$. We can frame one of these credits $\pounds 1$ to the postcondition, such that it becomes available when we need to prove $\mathtt{then}(p, f) \rightsquigarrow_{\mathrm{ghost}} ()$. That is, instead of $\boxed{\mathsf{TS}}^{\mathcal{N}} \vdash \mathtt{then}(p, f) \rightsquigarrow_{\mathrm{ghost}} ()$, we now prove $\boxed{\mathsf{TS}}^{\mathcal{N}} * \pounds 1 \vdash \mathtt{then}(p, f) \rightsquigarrow_{\mathrm{ghost}} ()$. Thus, when we open $\boxed{\mathsf{TS}}^{\mathcal{N}}$ this time, we can use the later credit $\pounds 1$ to eliminate the later and obtain $f(a) \rightsquigarrow_{\mathrm{ghost}} ()$, finishing the proof.

4 LATER CREDITS FOR LOGICAL ATOMICITY

In this section, we demonstrate another use of later credits, namely for eliminating a lingering pain point in one of Iris's specialties: *logical atomicity proofs* [Jung et al. 2015]. Inspired originally by the TaDA logic [da Rocha Pinto et al. 2014], logical atomicity is Iris's technique for proving functional correctness of (fine-grained) concurrent data structures. Akin to the standard notion of linearizability [Herlihy and Wing 1990], a logically atomic specification of a concurrent operation says that the operation *appears* to take effect atomically, even though it may actually take multiple physical steps. As a consequence, clients can reason about logically atomic operations (almost) as if they were physically atomic instructions—in particular, they can open invariants around them.

Logical atomicity has been successfully applied to a variety of challenging concurrent data structures [Jung et al. 2015, 2020; Birkedal et al. 2021; Frumin et al. 2021; Carbonneaux et al. 2022]. Unfortunately, in verifying logical atomicity for data structures that exhibit a common pattern known as "helping", step-indexing has always caused trouble. "Helping" refers to the situation where one thread helps another thread complete its operation. In previous work, proving logical atomicity for data structures with helping necessitated the use of an "ugly" workaround (to quote its inventor [Jung 2019]) called "make-laterable", which made logical atomicity harder to prove and harder to use for clients.

With later credits, we can avoid the need for "make-laterable" entirely, along with its limitations. To explain how, we will use a concrete example of a concurrent data structure that involves helping:

[6] One may wonder if we can get $f(a) \rightsquigarrow_{\mathrm{ghost}} ()$ out of the invariant during execution of $\mathtt{then}(p, f)$ in the program, so that there are still program steps around. The answer is no, because during that execution, the promise will still be in state **A**. Thus, f will be in the list, not executed, and $f(a) \rightsquigarrow_{\mathrm{ghost}} ()$ is not yet available, since it only enters the invariant in state **C**.

Implementation

$$\begin{aligned}
\texttt{new}() &\triangleq \textbf{let}\ (b,p) := (\textbf{ref}(0), \textbf{ref}(0)); \textbf{fork}\ \{\texttt{bg\_thread}(b,p)\}\,; (b,p)\\
\texttt{incr}(b,p) &\triangleq \textbf{let}\ n = \textbf{FAA}(p,1); \texttt{await\_backup}(b, n+1); n\\
\texttt{get}(b,p) &\triangleq \textbf{let}\ n = !\,p; \texttt{await\_backup}(b,n); n\\
\texttt{get\_backup}(b,p) &\triangleq\ !\,b
\end{aligned}$$

Helper Functions

$$\begin{aligned}
\texttt{bg\_thread}(b,p) &\triangleq \textbf{let}\ n = !\,p; b \leftarrow n; \texttt{bg\_thread}(b,p) \quad \text{// copy primary to backup, in a loop}\\
\texttt{await\_backup}(b,n) &\triangleq \textbf{if}\ !\,b < n\ \textbf{then}\ \texttt{await\_backup}(b,n)\ \textbf{else}\ () \quad \text{// loop until } !\,b \text{ reaches } n
\end{aligned}$$

Specification

$$\begin{aligned}
&\vdash \{\mathsf{True}\}\ \texttt{new}()\ \{c.\ \exists\gamma.\ \mathsf{is\_counter}^{\mathcal{N}}_{\gamma}(c) * \mathsf{value}_{\gamma}(0)\}\\
\mathsf{is\_counter}^{\mathcal{N}}_{\gamma}(c) &\vdash \langle n.\ \mathsf{value}_{\gamma}(n)\rangle\ \texttt{incr}(c)\ \langle m.\ m = n * \mathsf{value}_{\gamma}(n+1)\rangle_{\mathcal{N}}\\
\mathsf{is\_counter}^{\mathcal{N}}_{\gamma}(c) &\vdash \langle n.\ \mathsf{value}_{\gamma}(n)\rangle\ \texttt{get}(c)\ \langle m.\ m = n * \mathsf{value}_{\gamma}(n)\rangle_{\mathcal{N}}\\
\mathsf{is\_counter}^{\mathcal{N}}_{\gamma}(c) &\vdash \langle n.\ \mathsf{value}_{\gamma}(n)\rangle\ \texttt{get\_backup}(c)\ \langle m.\ m = n * \mathsf{value}_{\gamma}(n)\rangle_{\mathcal{N}}
\end{aligned}$$

Fig. 6. Counter with a backup.

a counter with a backup. We explain the counter (in §4.1), the intuitive argument for proving its logical atomicity (in §4.2), the reason we cannot implement that intuitive proof argument with "make-laterable" (in §4.3), and finally how later credits save the day (in §4.4).

4.1 A Counter with a Backup

Our motivating example is a counter with a backup (Figure 6). This counter is basically a regular monotone counter (as described in §2.1) with methods `incr` to increment the counter by 1 and `get` to get the current value of the counter. But there is a twist: the value of the counter is stored in two locations—the *primary* p and the *backup* b—and these two can get out of sync: the operation `incr` eagerly updates the primary p, but leaves updating the backup to *a background thread* (here, `bg_thread`). Clients can directly access the backup b through a third operation, `get_backup`, so it may seem like they can observe the difference between the primary and the backup. What makes this counter interesting is that they *cannot*, because `incr` and `get` wait for the backup to catch up.

To understand this counter better, let us consider a concrete example:

$$e_{\text{count}} \triangleq \textbf{let}\ c = \texttt{new}(); \textbf{fork}\ \{\texttt{incr}(c)\}\,; \textbf{let}\ x = \texttt{get}(c); \textbf{let}\ y = \texttt{get\_backup}(c); (x,y)$$

Depending on when the increment occurs and when the background thread updates the backup b, this expression has three possible values: $(0,0)$, $(0,1)$, and $(1,1)$. One value that it does not have is $(1,0)$. The outcome $(1,0)$ is impossible, even though `get` reads the primary p and `get_backup` reads the backup b. The reason is that `get` *waits* for the backup to catch up before returning its result, so we can be sure that any subsequent `get_backup` cannot read "outdated" values.

The counter with a backup is clearly a contrived example. However, it originates from an issue arising in real data structures that need to be durable. For example, a key-value server will store the current mapping of keys to values on disk, but also keep an in-memory copy of that mapping to quickly reply to read requests. Updating the data on disk is inefficient, so a background thread batches concurrent writes to be able to write them to disk in one go. At any time, the system can crash and the in-memory copy disappears; after reboot and recovery, the state of the key-value server is restored from what was stored on disk at the time of the crash. Since the in-memory copy can be lost, the operations working on it need to wait for their changes to become permanent, so they avoid returning data that is later lost in a crash. To avoid all the complexities of crashes and durable state, we have condensed this problem down to its core. The key-value store is replaced by a single counter, the durable disk is replaced by a second copy of the counter in memory, and we use `get_backup` to model the fact that this second copy is observable by clients through crashes.

Helping. The most interesting thing about the counter with a backup is the interaction between the background thread and the operations `get` and `incr`. Take `incr` for example. `incr` modifies the primary p, but its effect only becomes observable (through the counter operations) once the background thread updates the backup b. In other words, to complete its action, `incr` needs assistance from the background thread, which is typically called "helping". In general, helping means that the point in time when the action of one operation appears to clients to "take effect"—also known as the *linearization point* of the operation—is actually performed by *another* "helping" thread. For `incr`, the linearization point is the update of b to n in the background thread, because that is when the new counter value actually becomes observable to other `get` and `get_backup` operations. For `get`, in cases where the operation observes the primary p to be larger than the backup b, the linearization point is *also* the update of b in the background thread, because only then can other `get` and `get_backup` operations also observe the new value (see the e_{count} example).

What is particularly interesting about the helping in this example—and what makes it challenging to verify in existing Iris—is that at the point when the background thread updates b, it may have to help (an arbitrary number of) `get` operations complete, but it does not know which operations those are in advance because the `get` operations do nothing to explicitly communicate their need to be helped. In fact, the background thread may have to help `get` operations which only began *immediately before* the update of b to n. We call this phenomenon *unsolicited* helping, in contrast to the *solicited* helping that occurs in the `incr` operation (since the latter communicates explicitly to the background thread by incrementing p). As we will soon see, helping (especially unsolicited helping) makes it difficult to verify data structures like this one in existing Iris, but later credits offer a simpler way.

4.2 Logical Atomicity

Let us attempt to verify the counter. We want to prove the standard specification of a logically atomic counter, meaning `get` (and `get_backup`) observe the value of the counter at the linearization point and return it, while `incr` increments the value at the linearization point and returns the old value. In the language of logical atomicity, we express this with the specification shown in Figure 6. Except for the initialization (where atomicity does not matter), the specification consists of several *logically atomic triples* $\langle x.\, P\rangle\ f(a)\ \langle y.\, Q\rangle_{\mathcal{E}}$. These are special Hoare triples that describe the atomic action of f at the linearization point. For example, for `incr`, the logically atomic specification is

$$\big\langle n.\ \mathsf{value}_\gamma(n)\big\rangle\ \mathtt{incr}(c)\ \big\langle m.\ m = n * \mathsf{value}_\gamma(n+1)\big\rangle_{\mathcal{N}}$$

indicating that `incr` updates the value of the counter (identified by the logical name γ) from n to $n + 1$. Here, the number n is supposed to be the value of the counter *at the linearization point.* Since the number n is typically not known before the execution of `incr` (and potentially changes during its execution), logically atomic triples have an additional binder "$n.$" in their precondition. This binder can relate the value of n at the linearization point to the result of the triple "$m.$" in the postcondition. In the case of `incr`, we thus know that it returns the value of the counter at the linearization point similar to a fetch-and-add. The rest of the specification is bookkeeping: we keep some state of the counter in an invariant $\mathsf{is\_counter}^{\mathcal{N}}_{\gamma}(c)$, which means that the proof of `incr` needs access to invariant namespace $\mathcal{N}$ and that is reflected in the specification (it means clients may *not* open those invariants around a call to `incr`). We express the value of the counter with the (non-duplicable) predicate $\mathsf{value}_\gamma(n)$, and we connect both pieces through the name γ.

Logically atomic triples. Fully explaining how one proves and uses logically atomic specifications is beyond the space limitations of this paper. Fortunately, to understand the step-indexing troubles that arise, a deep understanding of logical atomicity is not required. It suffices to know a little bit more about the definition of a logically atomic triple (we are also more explicit about free variables here):

$$\langle x.\, P(x)\rangle\; e\; \langle y.\, Q(x,y)\rangle_{\mathcal{E}} \triangleq \forall R.\, \left\{\mathbf{AU}(x.\, P(x), y.\, Q(x,y))^{\mathcal{E}}_{R}\right\}\; e\; \{y.\, R(y)\}$$

Logically atomic triples are ordinary Hoare triples with a special *atomic update* $\mathbf{AU}$ in their precondition. The atomic update describes the "atomic action" of the operation. For example, in the case of `incr`, the atomic update would be $\mathbf{AU}_{\mathrm{inc}}(R) \triangleq \mathbf{AU}(n.\, \mathsf{value}_\gamma(n), m.\, m = n * \mathsf{value}_\gamma(n+1))^{\mathcal{N}}_{R}$, describing the abstract state change that we want `incr` to perform.

When we prove a logically atomic triple, it is our job to make sure the atomic update $\mathbf{AU}$ is executed, meaning we have to update the program state and ghost state atomically in the way described by the abstract action. We can update our state *atomically* by either (1) performing a physically atomic operation that corresponds to the update or (2) by calling another logically atomic operation. Executing the atomic update yields the "result" $R(y)$ in exchange. The fact that *we have to execute the update* is encoded somewhat implicitly: to prove the triple $\left\{\mathbf{AU}(x.\, P(x), y.\, Q(x,y))^{\mathcal{E}}_{R}\right\}\; e\; \{y.\, R(y)\}$, we eventually have to establish the postcondition $R(y)$. The postcondition R is *universally quantified* and the only way to obtain ownership of the result $R(y)$ is executing the atomic update.

Proving logical atomicity in the presence of helping. Let us return to helping. In the world of logical atomicity, helping means the helpee (*e.g.,* `incr`) transfers its atomic update to the helper (*e.g.,* the background thread). The helper then executes the atomic update at the linearization point of the helpee and returns the result (*e.g.,* $R(n)$). We refer to this mechanism as the *helping exchange.*

To understand the helping exchange better, we discuss helping the `incr` operation. We start with an *idealized* version of the exchange, because step-indexing sadly makes the matter more complicated. To initiate the exchange, `incr` sets up the following invariant:

$$I_{\mathrm{inc}}(n, R) \triangleq \boxed{(\mathbf{AU}_{\mathrm{inc}}(R) * \mathsf{pending}) \lor (R(n) * \mathsf{executed}) \lor \mathsf{acknowledged}}^{\mathcal{N}}$$

and shares it with the background thread through the invariant behind $\mathsf{is\_counter}^{\mathcal{N}}_{\gamma}(c)$. Here, we use the propositions pending, executed, and acknowledged to distinguish the different stages of the helping exchange.[7] Initially, in the pending stage, `incr` stores its update in I_{inc} and then waits for the background thread. The background thread eventually reads p and then updates b. In the step where it updates b, it linearizes the pending increment `incr`. To do so, it opens the invariant I_{inc}, takes out the atomic update $\mathbf{AU}_{\mathrm{inc}}$, executes it (similar to how one executes $\mathrel{|}\!\!\Rrightarrow P$), and puts the result $R(n)$ back into the invariant (advancing to the executed stage). Finally, `incr` observes the change of b, opens I_{inc}, and takes out the result $R(n)$ (advancing to the acknowledged stage).

4.3 Helping without Later Credits

Sadly, without later credits, the helping exchange for `incr` is more complicated, because step-indexing gets in the way. To execute $\mathbf{AU}_{\mathrm{inc}}$ in the background thread, we first have to obtain ownership of the atomic update, which means taking it out of the invariant I_{inc}. Unfortunately, there are some hurdles: $\mathbf{AU}_{\mathrm{inc}}$ is not timeless and is stored in an invariant (*i.e.,* I_{inc}), which itself is

[7] We keep pending, executed, and acknowledged abstract here to simplify the presentation. Internally, they use ghost state to encode the different stages of the helping mechanism.

stored in another invariant (*i.e.,* $\text{is\_counter}^{\mathcal{N}}_{\gamma}(c)$)—a step-indexing nightmare. So we cannot just take $\mathbf{AU}_{\text{inc}}$ out of the invariant, execute it, and put $R(n)$ back into the invariant, all in one step.

To escape this nightmare, the typical solicited helping proof is a play in three acts; we use `incr` to illustrate it. In the first act, the helper discovers the helpees it is helping (*e.g.,* through reading p). At this point, it gains access to a *witness* $\triangleright W_{\text{inc}}$ for the waiting helpee—a resource relevant for the helping exchange, but guarded by a "$\triangleright$". (It is not so important here what this witness is, only that we need to access it early.) In the second act, the helper takes a bookkeeping step of execution to eliminate the later (*e.g.,* the reduction of **`let`**). In the third act, the helper reaches the linearization point (*e.g.,* the update of b), and uses the witness W_{inc} to obtain access to $\mathbf{AU}_{\text{inc}}$ and execute it.

This strategy is suboptimal for several reasons. First, the helping exchange is more complicated than the intuition that we previously outlined and requires additional foresight. Second, making the dance with the witness W_{inc} work requires delicate step-indexing tricks behind the scenes. These tricks, known as "make-laterable", are so cumbersome that even its inventor called them "ugly" [Jung 2019]. (With later credits, "make-laterable" becomes obsolete, so we spare the reader the details here.) Third, "make-laterable" comes at a cost: clients of logically atomic specifications are faced with additional proof obligations when they want to use them. And last but not least, this strategy does not work for *unsolicited* helping: we now use `get` to illustrate why not.

When the background thread reads the primary p, it cannot gain access to the witnesses W_{get} of all the `get` operations it linearizes. The reason is that they might not be be there yet. That is, after the read of p, a new `get` operation could arrive. This `get` will be linearized with the update to b, but the background thread could not observe W_{get} yet when it read p. Thus, even with the established bag of tricks, we cannot realize the standard three-part play for `get`.

4.4 Helping with Later Credits

Enter later credits. With later credits, there is no need for a complicated three-act play because we can instead just eliminate the requisite number of laters right at the linearization point. Thus, we can avoid relying on the ugly "make-laterable" trick in the definition of $\mathbf{AU}$ (which in turn means fewer proof obligations for clients of logically atomic triples), and we can implement the idealized helping exchange as originally envisioned.

To enable helping proofs, we set up the following scheme with later credits: if a helpee wants help from a helper, it pays the helper with *a later credit* £1, which is sent along with the atomic update $\mathbf{AU}$ in the shared invariant. The credit remains in the invariant while the update is pending, and can be removed when the atomic update has been executed. For example, in the troubling case of `get`, the invariant becomes:

$$I_{\text{get}}(n,R) \triangleq \boxed{(\mathbf{AU}_{\text{get}}(R) * £1 * \text{pending}) \vee (R(n) * \text{executed}) \vee \text{acknowledged}}^{\mathcal{N}}$$

where $\mathbf{AU}_{\text{get}}(R) \triangleq \mathbf{AU}(n.\ \text{value}_{\gamma}(n), m.\ m = n * \text{value}_{\gamma}(n))^{\mathcal{N}}_{R}$ is the atomic update of `get`.

And that is it! In the presence of later credits, the idealized helping exchange just works. The later troubles vanish, since the helper always has a credit in hand when it needs to access an atomic update. For example, if the background thread needs to access the atomic update $\mathbf{AU}_{\text{get}}$, then it can use the later credit stored along side with $\mathbf{AU}_{\text{get}}$ to eliminate a guarding later from $\mathbf{AU}_{\text{get}}$. Afterwards, it can execute the atomic update and return the result $R(n)$. Since the update is no longer pending, it does not have to put any credits back.

For the helpee, producing the later credit is straightforward. In a non-trivial logically atomic operation, there are plenty of bookkeeping steps around that have nothing to do with the linearization point (*e.g.,* the first step of beta reduction, **`let`** bindings, arithmetic, etc.), which all generate credits (see PureStep). Since each of these steps generates a credit, but there is only one linearization point per operation, there are typically plenty of credits available.

To validate this point, we used later credits to reprove the major benchmarks for logical atomicity (*e.g.,* the elimination stack [Jung et al. 2015]) with our simplified definition of **AU**, replacing the three-act play by the idealized helping exchange. Make-laterable, be gone!

5 SOUNDNESS OF LATER CREDITS

We have now seen several applications of later credits, but we have yet to discuss how to justify their soundness. When we add later credits, the main challenge is that we have to ensure that the program logic remains *adequate* (*i.e.,* correctness proofs in Iris mean something outside the logic):

Theorem 5.1 (Adequacy). *Let ϕ be a first-order predicate on values. If $\vdash \{\mathsf{True}\}\ e\ \{v.\ \phi(v)\}$ and e executes in n steps to e', then e' can either take another step or e' is a value v and $\phi(v)$ is true.*

The adequacy theorem turns proofs of Hoare triples $\vdash \{\mathsf{True}\}\ e\ \{v.\ \phi(v)\}$ into a correctness property of e, namely that e is safe to execute and only terminates in values satisfying ϕ. The adequacy theorem in the presence of later credits will be the main result of this section. We explain how it was proved before (in §5.1), and why it remains sound even with later credits (in §5.2).

5.1 Model of Iris's Weakest Preconditions without Later Credits

To prove adequacy, we need to know how Iris models Hoare triples. In Iris, Hoare triples are defined as $\{P\}\ e\ \{v.\ Q\} \triangleq \Box(P \mathrel{-\!\!*} \mathsf{wp}\ e\ \{v.\ Q\})$, where $\mathsf{wp}\ e\ \{v.\ Q\}$ is the *weakest precondition* that is required for e to terminate safely in a value v satisfying Q or else diverge. Hence, to show a Hoare triple $\{P\}\ e\ \{v.\ Q\}$, we show that the precondition P implies (in the separation logic sense) the weakest precondition of e. (The persistence modality "$\Box$" ensures that Hoare triples are duplicable facts, so we can use them multiple times.)

The definition of $\mathsf{wp}\ e\ \{v.\ Q\}$ itself is quite a mouthful (see the supplementary material). To explain how later credits fit in, we condense it to its core by omitting masks and ignoring concurrency:

$$\mathsf{wp}\ e\ \{v.\ Q\} \triangleq {\mid\!\!\Rrightarrow}\ Q[e/v] \qquad \text{if } e \in \mathit{Val}$$

$$\begin{aligned}\mathsf{wp}\ e\ \{v.\ Q\} \triangleq \forall \sigma.\ \mathcal{S}(\sigma) \mathrel{-\!\!*} {\mid\!\!\Rrightarrow}\ &\mathsf{red}(e,\sigma) * \qquad \text{if } e \notin \mathit{Val}\\ &(\forall e', \sigma'.\ (e,\sigma) \to (e',\sigma') \mathrel{-\!\!*} \triangleright {\mid\!\!\Rrightarrow}\ \mathcal{S}(\sigma') * \mathsf{wp}\ e'\ \{v.\ Q\})\end{aligned}$$

In the value case, the weakest precondition is simply the postcondition Q after an update. Otherwise, if e is not a value, then the weakest precondition consists of a progress and a preservation part. (The proposition $\mathcal{S}$ is the *state interpretation* [Jung et al. 2018b, §7.3], which ties the heap σ in the weakest precondition to the assertions $\ell \mapsto v$ in the program logic. Since the heap is orthogonal to later credits, we will largely ignore the state interpretation in the following.) The progress part requires that e is never stuck in the current heap, meaning $\mathsf{red}(e,\sigma) \triangleq (\exists e', \sigma'.\ (e,\sigma) \to (e',\sigma'))$. The preservation part requires us to establish the weakest precondition again after every possible step of e.

When we prove a weakest precondition, the updates (${\mid\!\!\Rrightarrow}$) in the definition enable us to update our ghost state (see UpdExec) and the later ($\triangleright$) enables us to eliminate laters when we take program steps (see PureStep). Readers familiar with step-indexing may think of this later modality as a decrease in the underlying step-index. For example, if we prove $\mathsf{wp}\ e\ \{v.\ Q\}$ and we take a pure step $e \to_{\mathrm{pure}} e'$, then we only need to prove $\mathsf{wp}\ e'\ \{v.\ Q\}$ *later* (so one step-index lower), which justifies removing a guarding later from all assumptions in our current proof context. (Formally, the rule LaterMono is used to justify removing a later from both the goal and the assumptions.)

Proving adequacy. Let us now return to the adequacy theorem. To simplify the proof sketch, we prove a slightly weaker form (which ignores progress, because progress is proven analogously), and we start directly with the weakest precondition (instead of a Hoare triple):

Lemma 5.2. *Let* $\vdash \mathsf{wp}\ e_0\ \{v.\ \phi(v)\}$. *If* $(e_0, \sigma_0) \rightarrow^n (e_n, \sigma_n)$ *where* e_n *is a value, then* $\phi(e_n)$ *holds.*

Proof Sketch. We initialize the state interpretation $\mathcal{S}$, obtaining $\vdash {\Rrightarrow}\mathcal{S}(\sigma_0) * \mathsf{wp}\ e_0\ \{v.\ \phi(v)\}$. Consider the case where $n > 0$ and $(e_0, \sigma_0) \rightarrow (e_1, \sigma_1) \rightarrow^{n-1} (e_n, \sigma_n)$. We unfold the weakest precondition and thus obtain:

$$\vdash {\Rrightarrow}\mathcal{S}(\sigma_0) * (\forall \sigma.\ \mathcal{S}(\sigma) \mathrel{-\!\!*} {\Rrightarrow}\mathsf{red}(e_0, \sigma) * (\forall e', \sigma'.\ (e_0, \sigma) \rightarrow (e', \sigma') \mathrel{-\!\!*} \triangleright {\Rrightarrow}\ \mathcal{S}(\sigma') * \mathsf{wp}\ e'\ \{v.\ \phi(v)\}))$$

Since we have the the state interpretation $\mathcal{S}$ and a step of e_0, we can instantiate the assumptions, drop $\mathsf{red}(e_0, \sigma)$, and obtain $\vdash {\Rrightarrow} \triangleright {\Rrightarrow}\ \mathcal{S}(\sigma_1) * \mathsf{wp}\ e_1\ \{v.\ \phi(v)\}$. Doing the same again for the next step under the modalities "${\Rrightarrow} \triangleright {\Rrightarrow}$" yields $\vdash ({\Rrightarrow} \triangleright {\Rrightarrow})^2\ \mathcal{S}(\sigma_2) * \mathsf{wp}\ e_2\ \{v.\ \phi(v)\}$. We can then inductively repeat this process until we reach the value e_n. At that point, we obtain $\vdash ({\Rrightarrow} \triangleright {\Rrightarrow})^n\ {\Rrightarrow}\phi(e_n)$ after dropping the state interpretation. Thus, the weakest precondition yields our desired postcondition $\phi(e_n)$, albeit under a number of updates and later modalities. (We arrive at the same conclusion for $n = 0$.) Finally, to obtain $\phi(e_n)$ without the updates and laters, we use Lemma 5.3 (below). □

In Iris, if one proves a first-order proposition ϕ under an interleaving of updates and laters, then ϕ must hold, since it neither depends on step-indexing nor on resources:

Lemma 5.3. *Let* ϕ *be a first-order proposition. If* $\vdash ({\Rrightarrow} \triangleright {\Rrightarrow})^n\ {\Rrightarrow}\phi$, *then* ϕ *holds.*

How exactly this lemma is proven does not really matter for later credits, so we will not digress here.[8] What is important for us is that (1) it holds and (2) we can use it to prove adequacy of Hoare triples. With Lemma 5.3, we conclude our review of the original adequacy proof in Iris.

5.2 Modeling Later Credits

With later credits, we introduce an additional layer in between program steps and laters. In the definition of the weakest precondition above, the later "$\triangleright$" enables later eliminations. It tightly couples them to program steps, since after each program step, we get to eliminate another later. Later credits relax this connection. With later credits, a credit £1 becomes available with every program step. We can spend it immediately to eliminate a later, or we can save it for another proof step. This "amortized" form of reasoning about later eliminations works, because all that matters for adequacy is that in an n-step execution at most n laters are eliminated. It matters less (as we will see below) *when* these laters are eliminated, so we delegate the responsibility to "track" later eliminations to a new modality, the later elimination update ${\Rrightarrow}_{\mathsf{le}}P$.

The weakest precondition. We redefine the weakest precondition (roughly) as:

$$\begin{aligned} &\mathsf{wp}\ e\ \{v.\ Q\} \triangleq {\Rrightarrow}_{\mathsf{le}}Q[e/v] && \text{if } e \in \mathit{Val} \\ &\mathsf{wp}\ e\ \{v.\ Q\} \triangleq \forall \sigma.\ \mathcal{S}(\sigma) \mathrel{-\!\!*} {\Rrightarrow}_{\mathsf{le}}\mathsf{red}(e, \sigma)\ * && \text{if } e \notin \mathit{Val} \\ &\qquad (\forall e', \sigma'.(e, \sigma) \rightarrow (e', \sigma') \mathrel{-\!\!*} \text{£}1 \mathrel{-\!\!*} {\Rrightarrow}_{\mathsf{le}}\mathcal{S}(\sigma') * \mathsf{wp}\ e'\ \{v.\ Q\}) \end{aligned}$$

The changes are highlighted in purple: we use later elimination updates instead of standard updates and we make a new credit available after every step. In this updated definition, the connection between program steps and later eliminations is relaxed, because physical steps yield later credits which can subsequently be used for later eliminations virtually anywhere in the the rest of a proof.

[8] Readers familiar with step-indexing can think of the proof of Lemma 5.3 as picking the step-index $n + 1$, which lets one obtain the first-order proposition ϕ at step-index 1, meaning ϕ must hold.

Adequacy. Let us return to adequacy. For simplicity, we focus again on the special case:

Lemma 5.4. *Let* $\vdash \mathrm{wp}\ e_0\ \{v.\ \phi(v)\}$. *If* $(e_0, \sigma_0) \to^n (e_n, \sigma_n)$ *where* e_n *is a value, then* $\phi(e_n)$ *holds.*

Proof Sketch. The proof starts virtually unchanged, meaning we unfold the weakest precondition n times and we instantiate the execution of e_0. We then obtain:

$$\vdash (\Rrightarrow_{\mathsf{le}}(\pounds 1 \mathrel{-\!\!*} \Rrightarrow_{\mathsf{le}}))^n \Rrightarrow_{\mathsf{le}} \phi(e_n)$$

Instead of updates and laters, here we iterate later elimination updates and credit assumptions "$\pounds 1 \mathrel{-\!\!*}$". We can pull all the credit assumptions out and obtain $\pounds n \vdash (\Rrightarrow_{\mathsf{le}} \Rrightarrow_{\mathsf{le}})^n \Rrightarrow_{\mathsf{le}} \phi(e_n)$. Using transitivity of "$\Rrightarrow_{\mathsf{le}}$", this can be simplified to $\pounds n \vdash \Rrightarrow_{\mathsf{le}} \phi(e_n)$. From here, the desired goal $\phi(e_n)$ follows by chaining soundness of the later elimination update Lemma 5.5 (below) and Lemma 5.3. □

In the proof of Lemma 5.4, we can see that later credits delegate the responsibility of later management to the later elimination update. The proof boils down to $\pounds n \vdash \Rrightarrow_{\mathsf{le}} \phi(e_n)$, and from there we can retrieve the "amortized" interleaving of laters through the following soundness lemma:

Lemma 5.5. *If* $\pounds n \vdash \Rrightarrow_{\mathsf{le}} \phi$, *then* $\vdash (\Rrightarrow \triangleright \Rrightarrow)^n \Rrightarrow \phi$.

In other words, the later elimination update aggregates all of our step-index decreases and ghost state updates. We first define it and then return to the proof of Lemma 5.5 below.

The later elimination update. The later elimination update can be defined using the existing connectives of Iris, so we never have to touch the underlying model. In fact, we have discussed almost all the pieces that are needed to define it. There is just one missing, *the credit supply* $\pounds_\bullet\, m$. The credit supply $\pounds_\bullet\, m$ is a piece of ghost state tracking the total number of available credits. That is, its value m is, at any time, the sum of all the credits $\pounds n$ distributed in the logic. Thus, it satisfies the following rules:

SupplyBound
$$\pounds_\bullet\, m * \pounds n \vdash m \geq n$$

SupplyDecr
$$\pounds_\bullet\, (n + m) * \pounds n \vdash \Rrightarrow \pounds_\bullet\, m$$

The rule SupplyBound ensures that $\pounds_\bullet\, m$ is an upper bound. The rule SupplyDecr allows us to decrement the supply by giving up some credits. Readers familiar with Iris's model of resources can think of the supply $\pounds_\bullet\, m$ and the credits $\pounds n$ as the elements of the resource algebra $\mathit{Auth}(\mathbb{N}, +)$.[9]

We now turn to the later elimination update. Based on what we have seen, $\Rrightarrow_{\mathsf{le}}$ must be a monad, must connect later credits to later eliminations, and must enable ghost state updates. We define:

$$\Rrightarrow_{\mathsf{le}} P \triangleq \forall n.\ \pounds_\bullet\, n \mathrel{-\!\!*} \Rrightarrow ((\pounds_\bullet\, n \ * \ P) \vee (\exists m < n.\ \pounds_\bullet\, m \ * \ \triangleright \Rrightarrow_{\mathsf{le}} P))$$

Let us break the definition into pieces. First, the definition quantifies over the current credit supply $\pounds_\bullet\, n$. As a consequence, when we prove $\Rrightarrow_{\mathsf{le}} P$, we can make use of the rules SupplyDecr and SupplyBound to (potentially) decrease the credit supply if we are willing to give up a later credit $\pounds 1$. Second, after an update to the ghost state, the later elimination update offers a choice: we can either (1) return the supply and prove P (turning it into a standard update), or (2) decrease the supply and correspondingly wrap the goal with a later, thereby enabling one later to be eliminated from any assumption in the context. (Note that the latter case is analogous to how $\triangleright$ was used to support later elimination in Iris's original definition of the weakest precondition.) Finally, the definition is recursive, so we can repeat both ghost state updates and later eliminations (if we have additional credits). This recursion is handled with Iris's guarded fixpoints [Jung et al. 2018b, §5.6].

Let us now return to the soundness statement of the later elimination update:

[9] To be precise, the later credits $\pounds n \triangleq \boxed{\circ n}^{\gamma_{\mathsf{lc}}}$ are the fragments and the supply $\pounds_\bullet\, m \triangleq \boxed{\bullet m}^{\gamma_{\mathsf{lc}}}$ is the authoritative element of the resource algebra $\mathit{Auth}(\mathbb{N}, +)$. Both pieces are connected through the ghost name γ_{lc}, which is chosen globally.

Lemma 5.5. *If* $\pounds n \vdash |\!\!\Rrightarrow_{\mathsf{le}} \phi$, *then* $\vdash (|\!\!\Rrightarrow \triangleright |\!\!\Rrightarrow)^n |\!\!\Rrightarrow \phi$.

Proof Sketch. We allocate $\pounds_\bullet n$ and $\pounds n$, meaning we obtain $\vdash |\!\!\Rrightarrow (\pounds_\bullet n * \pounds n)$. Using our assumption $\pounds n \vdash |\!\!\Rrightarrow_{\mathsf{le}} \phi$, we obtain $\vdash |\!\!\Rrightarrow (\pounds_\bullet n * |\!\!\Rrightarrow_{\mathsf{le}} \phi)$. Analogous to the adequacy proof of the weakest precondition, we then unroll the later elimination update. That is, after unfolding "$|\!\!\Rrightarrow_{\mathsf{le}}$", we have:

$$\vdash |\!\!\Rrightarrow \pounds_\bullet n * (\forall n.\, \pounds_\bullet n \mathrel{-\!\!*} |\!\!\Rrightarrow ((\pounds_\bullet n \ * \ \phi) \vee (\exists m < n.\, \pounds_\bullet m \ * \ \triangleright |\!\!\Rrightarrow_{\mathsf{le}} \phi)))$$

which can be simplified to $\vdash |\!\!\Rrightarrow |\!\!\Rrightarrow (\pounds_\bullet n \ * \ \phi) \vee (\exists m < n.\ \triangleright(\pounds_\bullet m \ * \ |\!\!\Rrightarrow_{\mathsf{le}} \phi))$. In the left branch of the disjunction, we are done (using UpdReturn, UpdBind, and LaterIntro). In the right branch, we are in a similar situation as before: we have a separating conjunction of the credit supply and a later elimination update (*i.e.,* $\pounds_\bullet m \ * \ |\!\!\Rrightarrow_{\mathsf{le}} \phi$). Thus, we can repeat the unfold-then-simplify step. Every time we consider the right branch, the credit supply decreases (*e.g.,* from n to some $m < n$) and, since n is finite, this decrease can happen at most n times. Consequently, after n unfold-then-simplify steps, we know the left branch must have been chosen. □

Backwards compatibility. In large parts of Iris, including the weakest precondition, we replace the standard update modality "$|\!\!\Rrightarrow$" with the later elimination update "$|\!\!\Rrightarrow_{\mathsf{le}}$" (see the supplementary material). Since $|\!\!\Rrightarrow_{\mathsf{le}}$ and $|\!\!\Rrightarrow$ satisfy nearly identical rules, the later credits mechanism is mostly backwards compatible. The only rules that the new update modality does not satisfy are interaction rules with Iris's "plainly modality" $\blacksquare P$. These rules were introduced by Timany et al. [2018] for a logical relation for Haskell's ST monad, but are rarely used elsewhere in the Iris ecosystem.

6 EXTENSIONS

In this section, we discuss prior techniques to extend step-indexing and explain how later credits complement them. We give an overview of the techniques in the following table:

	Number of laters that can be eliminated per program step	
Traditional step-indexing	$1 \to 1 \to 1 \to 1 \to 1 \ \ldots$	
Folklore extension	$k \to k \to k \to k \to k \ \ldots$	for fixed k upfront
Flexible step-indexing	$1 \to 2 \to 3 \to 4 \to 5 \ \ldots$	
Transfinite step-indexing	$n_1 \to n_2 \to n_3 \to n_4 \to n_5 \ldots$	for arbitrary n_i per step

Traditional step-indexing allows for the elimination of exactly one later per program step. It is folklore that this can be relaxed to k laters per program step. Jourdan [2021] shows that the number k does not have to be fixed upfront, but can depend on the program execution, so one can eliminate 1 later after the first step, 2 after the second, 3 after the third, *etc.* This extension of step-indexing uses *time receipts* [Mével et al. 2019] to keep track of the number of program steps. It is also possible to eliminate an *arbitrary* number of laters per step as shown by Svendsen et al. [2016] for step-indexed logical relations and Spies et al. [2021] for Iris. This requires a transfinitely step-indexed model, *i.e.,* one with ordinals instead of natural numbers as step indices.

In all of these techniques, later elimination remains coupled to program steps—*i.e.,* a later can only be eliminated if the goal is a weakest precondition. Later credits are fundamentally different, because they turn the right to eliminate a later into an ownable resource $\pounds 1$ that can be saved and used even when the goal is merely an update modality. This decoupling is crucial for proofs where there is no program in sight when a later needs to be eliminated (*e.g.,* the examples from §3 and §4).

That said, later credits can be *combined* with these techniques to unlock additional, interesting applications. We combine flexible step-indexing and later credits (in §6.1) and use them for two examples: prepaid invariants and reverse refinements. We then discuss the extension of later credits to transfinite step-indexing and point out trade-offs compared to flexible finite step-indexing (§6.2).

$$\textsc{ReceiptTimeless}\quad \text{timeless}(⧗ n) \qquad \textsc{PureStepReceipt}\quad \frac{\{P * \pounds 1 * ⧗ 1\}\ e_2\ \{v.\ Q\} \qquad e_1 \rightarrow_{\text{pure}} e_2}{\{P\}\ e_1\ \{v.\ Q\}} \qquad \textsc{ReceiptCreditsPost}\quad \frac{\{P\}\ e\ \{v.\ Q\} \qquad e \notin \mathit{Val}}{\{P * ⧗ n\}\ e\ \{v.\ Q * \pounds n * ⧗ n\}}$$

Fig. 7. The proof rules for later credits with flexible step-indexing.

6.1 Flexible Step-Indexing

Similar to Jourdan [2021] and Mével et al. [2019], we use time receipts to reflect the number of previous program steps into Iris. The rules of our extension are shown in Figure 7. Each execution step produces a *time receipt* $⧗ 1$ and a credit $\pounds 1$ (PureStepReceipt). The receipts can be used to generate later credits (ReceiptCreditsPost). That is, if we own n receipts $⧗ n$, then we can leverage these receipts to generate an additional n credits $\pounds n$ after the next step of execution. We now briefly sketch two applications of this extension. Their details can be found in the supplementary material.

Prepaid invariants. Prepaid invariants $\boxed{R}^{\mathcal{N}}_{\text{pre}} \triangleq \boxed{R * \pounds 1 * ⧗ 1}^{\mathcal{N}}$ store a later credit and a time receipt. Their rules are as follows:

$$\textsc{InvPreAlloc}\quad \frac{\left\{P * \boxed{R}^{\mathcal{N}}_{\text{pre}}\right\}\ e\ \left\{v.\ Q\right\}}{\{P * \pounds 1 * ⧗ 1 * \triangleright R\}\ e\ \{v.\ Q\}} \qquad \textsc{InvPreOpen}\quad \frac{\{R * P\}\ e\ \{v.\ R * Q\}_{\mathcal{E}\setminus\mathcal{N}} \qquad \mathcal{N} \subseteq \mathcal{E} \qquad e \text{ physically atomic}}{\left\{\boxed{R}^{\mathcal{N}}_{\text{pre}} * P\right\}\ e\ \left\{v.\ Q\right\}_{\mathcal{E}}}$$

Their distinguishing feature is that they can be opened around physically atomic instructions *without a guarding later* (InvPreOpen). The trick is that when we open the underlying invariant the later credit $\pounds 1$ can be used to eliminate the guarding later from R, and the time receipt $⧗ 1$ can be used to restore the later credit in the postcondition (with ReceiptCreditsPost). A consequence of this trick is that we need to provide a credit and receipt when allocating the invariant (InvPreAlloc). Since there is no "$\triangleright$" in InvPreOpen, we do not have to worry about later elimination when we work with, for example, nested invariants (see §1). However, we *cannot* open prepaid invariants around updates without a guarding later (see InvOpenUpd), because updates do not generate later credits.

Reverse refinements. Later credits with flexible step-indexing can also solve an issue with step-indexed logical relations described by Svendsen et al. [2016]. The problem they highlight involves proving a contextual equivalence $F\ e \equiv_{\text{ctx}} e : \tau$ for all expressions $e : \tau$ given a function $F : \tau \rightarrow \tau$. One strategy to show such an equivalence is to split proving the equivalence into the two contextual refinements $F\ e \leq_{\text{ctx}} e : \tau$ and $e \leq_{\text{ctx}} F\ e : \tau$. To prove these contextual refinements, one can prove the expressions logically refine each other, according to a step-indexed logical relation $\leq_{\log}$ (analogous to §3). In a logical refinement of the form $e_1 \leq_{\log} e_2 : \tau$, steps of e_1 allow elimination of laters. Thus, in the direction $F\ e \leq_{\log} e : \tau$, evaluating $F\ e$ takes steps that provide opportunities to eliminate laters. In the *reverse refinement* $e \leq_{\log} F\ e : \tau$, that is not the case—we need to prove $e \leq_{\log} F\ e : \tau$ for all $e : \tau$, meaning that e could be a value, which takes no steps.

Svendsen et al. use a *transfinite* step-indexed logical relation to address this. But transfinite step-indexed models come with trade-offs (see §6.2), so we show that later credits provide an alternate solution. We do this by extending ReLoC and proving the reverse refinement example of Svendsen et al., as well as a new and more difficult example involving concurrent memoization.

6.2 Transfinite Step-Indexing

When combining Transfinite Iris [Spies et al. 2021] with later credits, we obtain a rule that allows us to allocate an arbitrary number of credits in the postcondition of an expression:

$$\textsc{CreditsPost}\quad \frac{\{P\}\; e\; \{v.\, Q\} \qquad e \notin \mathit{Val}}{\{P\}\; e\; \{v.\, Q * £n\}}$$

At first glance, transfinite step-indexing may seem like a strict improvement over flexible step-indexing since it lets us obtain n credits in each step without the need for time receipts (*cf.* ReceiptCreditsPost). But there is a trade-off: As Spies et al. [2021] explain, the commuting rules LaterExists and LaterSep are not sound in a transfinite model. These rules are used widely in existing Iris proofs (*e.g.,* for logical atomicity), and it is not clear whether all of those proofs could be salvaged in Transfinite Iris. Conversely, the rule CreditsPost is unsound in a finitely step-indexed model—it allows us to prove {True} **skip** {_. False} as shown in the supplementary material [Spies et al. 2022]. Thus, advanced users of later credits have a choice: work in the flexible finite model with time receipts, or work in the transfinite model without the commuting rules.

7 RELATED WORK

Multiple later eliminations per step. Svendsen et al. [2016] and Jourdan [2021] have proposed techniques to generalize the traditional approach of "one-later-per-step" to allow multiple step-index decreases per step. We have discussed these techniques in §6. As explained there, later credits are not an alternative to these techniques, they *complement* them. In particular, for the applications presented in this work—especially those in §3 and §4—it is vital that we can eliminate later modalities even when we are not proving a Hoare triple, which neither approach supports.

Steel. Steel [Swamy et al. 2020; Fromherz et al. 2021] is a shallow embedding of concurrent separation logic in F*. Inspired by Iris, Steel supports dynamically allocated invariants but unlike Iris, opening an invariant in Steel does not introduce a later. Nevertheless, the underlying soundness argument crucially relies on program steps [Swamy et al. 2020, Page 18], as in Iris. The difference arises because Steel treats ghost operations such as opening invariants as *explicit ghost code* that can take steps (which can then be erased before execution), allowing them to hide the later modality from the rule to open invariants. The price for this more convenient interface is a loss in expressiveness—there is no Steel connective corresponding to Iris's update modality ("ghost actions without code", which logically atomic specifications are built on), and the authors of Steel say that "contextual refinement proofs are beyond what is possible in Steel" [Fromherz et al. 2021, Page 27].

Logical atomicity and linearizability. The main point of logical atomicity is to put user-defined, linearizable operations on (almost) the same footing as physically atomic instructions. In particular, users can open invariants around logically atomic operations. Prior work on logical atomicity either does not support helping [da Rocha Pinto et al. 2014] or relies on impredicative invariants in a step-indexed separation logic [Svendsen and Birkedal 2014; Jung et al. 2015, 2020] with its suite of later elimination challenges. Later credits thus represent a significant step forward for logical atomicity proofs in general.

As Birkedal et al. [2021] show, logical atomicity can also be used to prove the more traditional notion of linearizability [Herlihy and Wing 1990]. To express and prove linearizability, many alternative approaches have been studied in prior work [Elmas et al. 2010; Liang and Feng 2013; Turon et al. 2013; Sergey et al. 2015; Chakraborty et al. 2015; Khyzha et al. 2016; Nanevski et al. 2019]. Those alternatives do not rely on impredicative invariants and, hence, they do not suffer from the step-indexing problems that later credits help solve. Instead, they have other means of expressing and establishing linearizability, *e.g.*, specifications that expose the effects of linearizable operations using a PCM of event histories [Sergey et al. 2015; Nanevski et al. 2019]. What these approaches cannot do, compared to logical atomicity, is allow clients to treat linearizable operations "as if" they were physically atomic, meaning clients cannot open invariants around such user-defined operations, which is the main selling point of logical atomicity.

Reordering refinements. For stateful programming languages Benton *et al.* investigated reordering refinements based on type-and-effect systems in a series of papers [Benton et al. 2006; Benton and Buchlovsky 2007; Benton et al. 2007, 2009]. Their work was extended by Thamsborg and Birkedal [2011] to a language with higher-order state and effect-masking. We focus on the work of Krogh-Jespersen et al. [2017] (which generalizes Thamsborg and Birkedal [2011]) and Timany et al. [2018], because they consider languages with cyclic features such as higher-order state and recursive types. These cyclic features are typically what motivate the use of step-indexing in a logical relation, but they also add an additional layer of complexity in reordering proofs. Krogh-Jespersen et al. [2017] prove reorderings in a concurrent stateful language with an effect type system; their model supports reorderings of operations that write to *disjoint* parts of the heap and thus does not scale to the promise operations in §3. Timany et al. [2018] prove reorderings for a sequential language in the presence of Haskell's ST monad. They verify reorderings of pure computations (with stateful subcomputations encapsulated using ST), but for stateful computations they only show the expected monadic rules for the state monad. As mentioned earlier, our definition of reordering refinement is inspired by Timany et al. [2018]. The crucial difference is that we "bake in" support for later credits. As a consequence, we can prove the higher-order stateful reorderings of the promise operations (in §3), which are beyond the model of Timany et al. The key step in the proof, where we use an impredicative invariant to transfer the source execution between operations, is only possible because we can use later credits to eliminate the irksome laters that pop up.

ACKNOWLEDGMENTS

We wish to thank Jacques-Henri Jourdan and Yusuke Matsushita for feedback and helpful discussions, Upamanyu Sharma for supplying the original example that led to the counter with a backup, and the anonymous reviewers for their helpful feedback. This research was supported in part by a European Research Council (ERC) Consolidator Grant for the project "RustBelt", funded under the European Union's Horizon 2020 Framework Programme (grant agreement no. 683289), in part by the Dutch Research Council (NWO), project 016.Veni.192.259, in part by NSF award 2123864, and in part by generous gifts from Oracle Labs and Google.

REFERENCES

Amal Ahmed. 2004. *Semantics of types for mutable state.* Ph. D. Dissertation. Princeton University.

Amal Ahmed, Andrew W. Appel, Christopher D. Richards, Kedar N. Swadi, Gang Tan, and Daniel C. Wang. 2010. Semantic foundations for typed assembly languages. *TOPLAS* 32, 3 (2010), 1–67. https://doi.org/10.1145/1709093.1709094

Andrew W. Appel and David A. McAllester. 2001. An indexed model of recursive types for foundational proof-carrying code. *TOPLAS* 23, 5 (2001), 657–683. https://doi.org/10.1145/504709.504712

Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. 2007. A very modal model of a modern, major, general type system. In *POPL*. 109–122. https://doi.org/10.1145/1190216.1190235

Nick Benton and Peter Buchlovsky. 2007. Semantics of an effect analysis for exceptions. In *TLDI*. 15–26. https://doi.org/10.1145/1190315.1190320

Nick Benton, Andrew Kennedy, Lennart Beringer, and Martin Hofmann. 2007. Relational semantics for effect-based program transformations with dynamic allocation. In *PPDP*. 87–96. https://doi.org/10.1145/1273920.1273932

Nick Benton, Andrew Kennedy, Lennart Beringer, and Martin Hofmann. 2009. Relational semantics for effect-based program transformations: higher-order store. In *PPDP*. 301–312. https://doi.org/10.1145/1599410.1599447

Nick Benton, Andrew Kennedy, Martin Hofmann, and Lennart Beringer. 2006. Reading, writing and relations. In *APLAS*. LNCS, Vol. 4279. 114–130. https://doi.org/10.1007/11924661_7

Lars Birkedal, Thomas Dinsdale-Young, Armaël Guéneau, Guilhem Jaber, Kasper Svendsen, and Nikos Tzevelekos. 2021. Theorems for free from separation logic specifications. *PACMPL* 5, ICFP (2021), 1–29. https://doi.org/10.1145/3473586

Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang. 2011. Step-indexed Kripke models over recursive worlds. In *POPL*. 119–132. https://doi.org/10.1145/1926385.1926401

Stephen Brookes. 2007. A semantics for concurrent separation logic. *TCS* 375, 1-3 (2007), 227–270. https://doi.org/10.1016/j.tcs.2006.12.034

Alexandre Buisse, Lars Birkedal, and Kristian Støvring. 2011. Step-Indexed Kripke Model of Separation Logic for Storable Locks. In *MFPS (ENTCS, Vol. 276)*. 121–143. https://doi.org/10.1016/j.entcs.2011.09.018

Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. 2018. VST-Floyd: A separation logic tool to verify correctness of C programs. *JAR* 61, 1-4 (2018), 367–422. https://doi.org/10.1007/s10817-018-9457-5

Quentin Carbonneaux, Noam Zilberstein, Christoph Klee, Peter W. O'Hearn, and Francesco Zappa Nardelli. 2022. Applying formal verification to microkernel IPC at Meta. In *CPP*. 116–129. https://doi.org/10.1145/3497775.3503681

Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2019. Verifying concurrent, crash-safe systems with Perennial. In *SOSP*. 243–258. https://doi.org/10.1145/3341301.3359632

Soham Chakraborty, Thomas A. Henzinger, Ali Sezgin, and Viktor Vafeiadis. 2015. Aspect-oriented linearizability proofs. *LMCS* 11, 1 (2015). https://doi.org/10.2168/LMCS-11(1:20)2015

Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A logic for time and data abstraction. In *ECOOP (LNCS, Vol. 8586)*. 207–231. https://doi.org/10.1007/978-3-662-44202-9_9

Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2020. RustBelt meets relaxed memory. *PACMPL* 4, POPL (2020), 34:1–34:29. https://doi.org/10.1145/3371102

Derek Dreyer, Amal Ahmed, and Lars Birkedal. 2011. Logical Step-Indexed Logical Relations. *LMCS* 7, 2:16 (2011), 1–37. https://doi.org/10.2168/LMCS-7(2:16)2011

Tayfun Elmas, Shaz Qadeer, Ali Sezgin, Omer Subasi, and Serdar Tasiran. 2010. Simplifying Linearizability Proofs with Reduction and Abstraction. In *TACAS (LNCS, Vol. 6015)*. 296–311. https://doi.org/10.1007/978-3-642-12002-2_25

Aymeric Fromherz, Aseem Rastogi, Nikhil Swamy, Sydney Gibson, Guido Martínez, Denis Merigoux, and Tahina Ramananandro. 2021. Steel: proof-oriented programming in a dependently typed concurrent separation logic. *PACMPL* 5, ICFP, 1–30. https://doi.org/10.1145/3473590

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2018. ReLoC: A mechanised relational logic for fine-grained concurrency. In *LICS*. 442–451. https://doi.org/10.1145/3209108.3209174

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2021. ReLoC Reloaded: A Mechanized Relational Logic for Fine-Grained Concurrency and Logical Atomicity. *LMCS* 17, 3 (2021). https://doi.org/10.46298/lmcs-17(3:9)2021

Paolo G. Giarrusso, Léo Stefanesco, Amin Timany, Lars Birkedal, and Robbert Krebbers. 2020. Scala step-by-step: soundness for DOT with step-indexed logical relations in Iris. *PACMPL* 4, ICFP (2020), 114:1–114:29. https://doi.org/10.1145/3408996

Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: a correctness condition for concurrent objects. *TOPLAS* 12, 3 (1990), 463–492. https://doi.org/10.1145/78969.78972

Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2020. Actris: session-type based reasoning in separation logic. *PACMPL* 4, POPL (2020), 6:1–6:30. https://doi.org/10.1145/3371074

Jonas Kastberg Hinrichsen, Daniël Louwrink, Robbert Krebbers, and Jesper Bengtson. 2021. Machine-checked semantic session typing. In *CPP*. 178–198. https://doi.org/10.1145/3437992.3439914

Jacques-Henri Jourdan. 2021. Flexible number of logical steps per physical step. https://gitlab.mpi-sws.org/iris/iris/-/merge_requests/595 Iris merge request 595.

Ralf Jung. 2019. Logical Atomicity in Iris: The Good, the Bad, and the Ugly. https://people.mpi-sws.org/~jung/iris/logatom-talk-2019.pdf Presented at the Iris Workshop (https://iris-project.org/workshop-2019/).

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018a. RustBelt: Securing the foundations of the Rust programming language. *PACMPL* 2, POPL (2018), 66:1–66:34. https://doi.org/10.1145/3158154

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *ICFP*. 256–269. https://doi.org/10.1145/2951913.2951943

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

Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2020. The future is ours: prophecy variables in separation logic. *PACMPL* 4, POPL (2020), 45:1–45:32. https://doi.org/10.1145/3371113

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*. 637–650. https://doi.org/10.1145/2676726.2676980

Artem Khyzha, Alexey Gotsman, and Matthew J. Parkinson. 2016. A Generic Logic for Proving Linearizability. In *FM (LNCS, Vol. 9995)*. 426–443. https://doi.org/10.1007/978-3-319-48989-6_26

Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: A general, extensible modal framework for interactive proofs in separation logic. *PACMPL* 2, ICFP (2018), 77:1–77:30. https://doi.org/10.1145/3236772

Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017a. The essence of higher-order concurrent separation logic. In *ESOP (LNCS, Vol. 10201)*. 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017b. Interactive proofs in higher-order concurrent separation logic. In *POPL*. 205–217. https://doi.org/10.1145/3093333.3009855

Morten Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal. 2017. A relational model of types-and-effects in higher-order concurrent separation logic. In *POPL*. 218–231. https://doi.org/10.1145/3093333.3009877

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 *ESOP (LNCS, Vol. 12075)*. 336–365. https://doi.org/10.1007/978-3-030-44914-8_13

Hongjin Liang and Xinyu Feng. 2013. Modular verification of linearizability with non-fixed linearization points. In *PLDI*. 459–470. https://doi.org/10.1145/2491956.2462189

Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2019. Time credits and time receipts in Iris. In *ESOP (LNCS, Vol. 11423)*. 3–29. https://doi.org/10.1007/978-3-030-17184-1_1

Aleksandar Nanevski, Anindya Banerjee, Germán Andrés Delbianco, and Ignacio Fábregas. 2019. Specifying concurrent programs in separation logic: morphisms and simulations. *PACMPL* 3, OOPSLA (2019), 161:1–161:30. https://doi.org/10.1145/3360587

Peter W. O'Hearn. 2007. Resources, concurrency, and local reasoning. *TCS* 375, 1-3 (2007), 271–307. https://doi.org/10.1016/j.tcs.2006.12.035

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local reasoning about programs that alter data structures. In *CSL (LNCS, Vol. 2142)*. 1–19. https://doi.org/10.1007/3-540-44802-0_1

John C. Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *LICS*. 55–74. https://doi.org/10.1109/LICS.2002.1029817

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. Specifying and Verifying Concurrent Algorithms with Histories and Subjectivity. In *ESOP (LNCS, Vol. 9032)*. 333–358. https://doi.org/10.1007/978-3-662-46669-8_14

Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2021. Transfinite Iris: Resolving an existential dilemma of step-indexed separation logic. In *PLDI*. 80–95. https://doi.org/10.1145/3453483.3454031

Simon Spies, Lennard Gäher, Joseph Tassarotti, Ralf Jung, Robbert Rebbers, Lars Birkedal, and Derek Dreyer. 2022. Later credits Coq development and technical documentation. https://doi.org/10.5281/zenodo.6702804 Latest development at https://plv.mpi-sws.org/later-credits/.

Kasper Svendsen and Lars Birkedal. 2014. Impredicative concurrent abstract predicates. In *ESOP (LNCS, Vol. 8410)*. 149–168. https://doi.org/10.1007/978-3-642-54833-8_9

Kasper Svendsen, Filip Sieczkowski, and Lars Birkedal. 2016. Transfinite step-indexing: Decoupling concrete and logical steps. In *ESOP (LNCS, Vol. 9632)*. 727–751. https://doi.org/10.1007/978-3-662-49498-1_28

Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, and Guido Martínez. 2020. SteelCore: an extensible concurrent separation logic for effectful dependently typed programs. *PACMPL* 4, ICFP (2020), 121:1–121:30. https://doi.org/10.1145/3409003

Joseph Tassarotti, Ralf Jung, and Robert Harper. 2017. A higher-order logic for concurrent termination-preserving refinement. In *ESOP (LNCS, Vol. 10201)*. 909–936. https://doi.org/10.1007/978-3-662-54434-1_34

Jacob Thamsborg and Lars Birkedal. 2011. A Kripke logical relation for effect-based program transformations. In *ICFP*. 445–456. https://doi.org/10.1145/2034773.2034831

Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. 2018. A logical relation for monadic encapsulation of state: proving contextual equivalences in the presence of runST. *PACMPL* 2, POPL (2018), 64:1–64:28. https://doi.org/10.1145/3158152

Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*. 377–390. https://doi.org/10.1145/2500365.2500600

Hengchu Zhang, Wolf Honoré, Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. 2021. Verifying an HTTP Key-Value Server with Interaction Trees and VST. In *ITP (LIPIcs, Vol. 193)*. 32:1–32:19. https://doi.org/10.4230/LIPIcs.ITP.2021.32