Later Credits: Resourceful Reasoning for the Later Modality

SIMON SPIES, MPI-SWS, Germany
JOSEPH TASSAROTTI, Boston College, USA
RALF JUNG, MPI-SWS, Germany
LENNARD GÄHER, MPI-SWS, Germany
ROBBERT KREBBERS, Radboud University Nijmegen, The Netherlands
LARS BIRKEDAL, Aarhus University, Denmark
DEREK DREYER, MPI-SWS, 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 a powerful tool 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 guarded recursive 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 proof patterns which were previously not possible in Iris but are now possible due to later credits.

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. 2002], have since become an indispensable tool for building semantic models of modern type systems—such as those of Scala [Giarrusso et al. 2020] and Rust [Jung et al. 2018a]—which include “cyclic features” like recursive subtyping and higher-order state [Ahmed 2004; Birkedal et al. 2011]. Separation logic [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].

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 (e.g., “logical atomicity” [Jung et al. 2015]), building semantic models of higher-order, imperative, and concurrent languages (e.g., RustBelt [Jung et al. 2018a; Dang et al. 2020]), and deriving custom program logics for a variety of application domains (e.g., Actris [Hinrichsen et al. 2019], Aneris [Krogh-Jespersen et al. 2020], Perennial [Chajed et al. 2019], Key-Value Server [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]. Although it has been glossed over in much of the literature as an esoteric technical detail best left to the experts, the later modality is in fact central to how step-indexed separation logics work because 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 $n$ (with the intuitive meaning “$P$ 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 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 witnessed in earlier formulations of step-indexing [Dreyer et al. 2011].

However, in practice, particularly when working with more advanced step-indexed constructions, 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. But once $\triangleright P$ appears in a hypothesis, the name of the game is figuring out how to eliminate the guarding later modality in order to make use of the underlying 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—and concurrent separation logics in general—invariants $[R]$ 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 $\ell$ and, at the same time, constrain $\ell$ to only store natural numbers. Once the invariant $[R]$ 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):

$$
\begin{array}{cl}
\{ R \triangleright P \} e \{ R \triangleright Q \} & \text{e physically atomic} \\
[ R ] & \triangleright \{ P \} e \{ Q \}
\end{array}
$$

The rule says that if we open the invariant $[R]$, 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 $[R]$ 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 \textit{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 $R$ are not well-founded (i.e., inductive or co-inductive definitions do not suffice). To obtain a well-founded model of $R$, the only known approach is to stratify the cyclic construction using step-indexing [Svendsen and Birkedal 2014].

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 \textit{later modality}, which acts as a kind of “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:

\[
\text{InvOpen} \quad \begin{array}{c}
\{ \triangleright R \ast P \} e \{ v. \triangleright R \ast Q \}\downarrow N \\
N \subseteq E \\
e \text{ physically atomic}
\end{array}
\]

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 \textit{namespace} $N$, and Hoare triples with a \textit{mask} $E$. 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 \textit{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 for eliminating guarding laters:

1. \textit{Timeless propositions}. For the subclass of so-called \textit{timeless} propositions—which include propositions that are pure (e.g., \textit{even}(n)) or assert only first-order ownership (e.g., $\ell \mapsto \rightarrow 42$)—laters can be eliminated because the model of these propositions ignores the step-index.

2. \textit{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. \textit{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 \textit{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: \textit{nested invariants}. Consider the invariant:

\[
\exists \ell. \exists n : N. \ell \mapsto n + N_i \ast y \mapsto \text{ghost } \ell
\]

Here, the location $\ell$ is existentially quantified and connected to a logical identifier $y$ through a “ghost link” $y \mapsto \text{ghost } \ell$ (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 ($\exists n : N. \ell \mapsto n$) to justify the next step, then we are in a quandary. If we open the outer invariant, we get $\triangleright (\exists \ell. \exists n : N. \ell \mapsto n + N_i \ast y \mapsto \text{ghost } \ell)$. After applying commuting and timelessness rules, and eliminating the existential, we are left with

\[\text{We use nested invariants here, because they are one of the simpler examples to illustrate where the existing practices are not enough. In practice, most proofs do not need to use nested invariants. However, plenty of proofs put other (more complicated) step-indexed assertions into invariants, and then guarding laters can be a deal breaker (e.g., see §3 and §4).} \]
At this point, we have a later guarding \( \exists n : \mathbb{N} . \ell \mapsto n \)^{N_i} 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, and at present the only way of handling such scenarios is to do some non-trivial, non-local refactoring of the proof structure. In our experience, these scenarios are particularly frustrating because it is difficult to give users any clear intuition as to why a different proof structure is really required.

**Later credits.** In this paper, we present a fourth option for later elimination—*later credits*—which enables users to avoid non-local proof refactoring by exploiting the fact that we are working in a separation logic. 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 \( £ \) 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. In particular, it can be used as part of purely logical reasoning where there is no program in sight. For example, we can save a credit \( £ \) from one step, keep it for two subsequent steps, and the use it before the next step to eliminate a later guarding an invariant assertion.

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” \( £ \) (e.g., 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.

The reasoning that later credits enable has two kinds of applications: First, it can simplify existing proofs. It can help where step-indexing previously got in the way and cluttered the proofs. For example, with later credits we can define a new kind of invariant, a *prepaid invariant* \( [R^N_{pre}] \), which can be opened around atomic expressions *without* a guarding later:

\[
\text{INVPREOPEN}
\begin{align*}
\{ R \ast P \} e \{ v. R \ast Q \}_{E \setminus N} & \quad N \subseteq \mathbb{E} \quad e \text{ physically atomic} \\
[R^N_{pre}] + \{ P \} e \{ v. Q \}_E
\end{align*}
\]

The crux of prepaid invariants is that the later elimination has been prepaid—accounted for in amortized fashion by an earlier step of computation. Second, later credits can enable proofs which were seemingly not possible with the standard later elimination techniques. For example, we will look at a new class of “helping” patterns in logical atomicity proofs which fundamentally rely on the reasoning provided by later credits.
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 intuitively explain later credits (in §2) and demonstrate how they work using the example of prepaid invariants (in §2.3). We explain their model and its soundness (in §5) and we discuss their limitations (in §6). We demonstrate the use of later credits with two applications:

1. To prove contextual refinement of two programs, one well-established proof technique is to use a step-indexed logical relation. However, as Svendsen et al. [2016] have pointed out, there are challenging cases where traditional step-indexed relation techniques fall short. For those cases, Svendsen et al. propose a transfinite step-indexed logical relation. In this work (in §3), we show how to use later credits to handle these cases without having to use a transfinitely step-indexed model (which has its own drawbacks, as pointed out in §6 and §7).

2. Verifying concurrent algorithms with helping is challenging because one thread is relying on another to be able to complete its operation. In Iris, the gold standard to verify these algorithms is logical atomicity. Proving logical atomicity of helping algorithms crucially relies on impredicativity of Iris’s invariants, which poses nontrivial problems as far as step-indexing is concerned. Earlier work used a clever hack to complete these proofs. In this work, we look at a challenging helping pattern we dub “unsolicited helping”, for which the clever hack appears insufficient, but can be verified with later credits as we show in §4.

We mechanized later credits [Author(s) 2021] and all of the above examples, including prepaid invariants, in the Coq proof assistant using the Iris Proof Mode [Krebbers et al. 2017b, 2018].

2 LATER CREDITS IN A NUTSHELL

In this section, we explain the key ideas behind later credits in Iris (in §2.2) and demonstrate how they are used with the example of prepaid invariants (in §2.3). Before we do either, we first take a step back and 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 $\{P\} e \{v. Q\}$. Similar to traditional Hoare logic, the Hoare triple $\{P\} e \{v. Q\}$ expresses that under precondition $P$ the expression $e$ terminates in a value $v$ satisfying postcondition $Q$, or $e$ diverges. 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 $\ell$ currently stores $42$ in the heap, but also that no other program part can modify $\ell$ while it owns $\ell \mapsto 42$. For example, in separation logic it is trivial to prove

$$\begin{align*}
\{\text{True}\} f () & \{v. \text{True}\} \\
\{r \mapsto 0\} r & \leftarrow 42; f (); !r \{v. v = 42\}
\end{align*}$$

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 $f ()$. 

With the concurrency primitive FAA will be a vital ingredient in the definition of later credits. To understand how they work, we consider an example: a fine-grained concurrent (implemented as a reference internally), we can read its value with `get` 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 `mk` only increases in value). As depicted in Figure 2, our counter offers three methods: we can create a counter with `mk_counter` (implemented with a concurrency primitive FAA, which does an atomic fetch-and-add and returns the old value). 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 [Calcagno et al. 2007]: state that is not physically present in the program but useful for its verification. Resources will be a vital ingredient in the definition of later credits. 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 counter `(c, n)` which expresses that the value of counter `c` is currently at least `n`. Importantly, 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

---

**Separation Logic:**

<table>
<thead>
<tr>
<th>Frame</th>
<th>Implementation Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{P} e {v. Q}</code></td>
<td><code>mk_counter () ≜ ref(0)</code></td>
</tr>
<tr>
<td><code>{P * R} e {v. Q * R}</code></td>
<td><code>{True} mk_counter () {c. counter(c, 0)}</code></td>
</tr>
<tr>
<td><code>{P * R} e {v. Q * R}</code></td>
<td><code>get(c) ≜ !c</code></td>
</tr>
<tr>
<td><code>{P +} e {R * Q}</code></td>
<td><code>{counter(c, n)} get(c) {m ∈ ℕ. counter(c, m) ∧ m ≥ n}</code></td>
</tr>
<tr>
<td><code>{P +} e {R * Q}</code></td>
<td><code>inc(c) ≜ FAA(c, 1)</code></td>
</tr>
<tr>
<td><code>{P +} e {R * Q}</code></td>
<td><code>{counter(c, n)} inc(c) {m ∈ ℕ. counter(c, m + 1) ∧ m ≥ n}</code></td>
</tr>
</tbody>
</table>

---

**Step-Indexing:**

- **LaterIntro**
  
  \[
  P + \rightarrow P
  \]

- **LaterMono**
  
  \[
  P + \quad (P +) + \rightarrow P
  \]

- **PointsToTimeless**
  
  \[
  \text{timeless}(t \mapsto v)
  \]

- **InvAlloc**
  
  \[
  \text{inv} \;\text{alloc}(R^N) + \{P\} e \{w. Q\}
  \]

- **InvOpen**
  
  \[
  \text{inv-open}(\text{open} \{\text{init} \;\text{P} e \{v. r \} \}_{\mathcal{E} \cap \mathcal{N}} \quad \mathcal{N} \subseteq \mathcal{E}
  \]

- **LaterExists**
  
  \[
  X \quad \text{non-empty}
  \]

- **PureStep**
  
  \[
  \text{pure}(e_1) \rightarrow e_2
  \]

- **LaterSep**
  
  \[
  \text{later-sep}(\text{P} + \rightarrow \text{P} + \rightarrow \text{Q})
  \]

- **LaterMono**
  
  \[
  \text{later-mono}(\text{P} + \rightarrow \text{P} + \rightarrow \text{Q})
  \]

- **PureStep**
  
  \[
  \text{pure}(e_1) \rightarrow e_2
  \]

- **LaterIntro**
  
  \[
  P + \rightarrow P
  \]

- **LaterMono**
  
  \[
  P + \rightarrow P
  \]

- **PointsToTimeless**
  
  \[
  \text{timeless}(t \mapsto v)
  \]

---

**Fig. 1.** A selection of Iris’s proof rules.

**Fig. 2.** Implementation and specification of a concurrent monotone counter.
about lower bounds). We define the counter predicate with an invariant and resources:

\[
\text{counter}(c, n) \triangleq \exists \gamma. \left[ \exists m : \mathbb{N}. c \mapsto m \cdot \text{mono}_\gamma(m) \right] \gamma N \cdot \text{lb}_\gamma(n)
\]

We use the invariant \(\left[ \exists m : \mathbb{N}. c \mapsto m \cdot \text{mono}_\gamma(m) \right] \gamma N\) to share ownership of \(c \mapsto n\) between threads. In contrast to the points-to assertion \(c \mapsto n\), invariants can be duplicated and, hence, shared with other threads. Thus, each thread that knows about \(\text{counter}(c, n)\) can open the invariant for an atomic step to get ownership of \(c \mapsto m\) and \(\text{mono}_\gamma(m)\), and has to return both after the step. We already know the assertion \(c \mapsto m\) conveys ownership of the location \(c\), in this case storing the counter value \(m\). The assertion \(\text{mono}_\gamma(m)\) is an exclusively owned resource with name \(\gamma\). 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 \(\text{lb}_\gamma(n)\), part of the definition of \(\text{counter}(c, n)\), is another resource. It is a lower bound on the monotonically growing resource \(\text{mono}_\gamma(m)\). Thus, if we own \(\text{mono}_\gamma(m)\) and \(\text{lb}_\gamma(n)\), then we can deduce \(m \geq n\) (i.e., \(\text{mono}_\gamma(m) \cdot \text{lb}_\gamma(n) \vdash m \geq n\)). 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 \(\text{mono}_\gamma(n)\) to \(\text{mono}_\gamma(n + 1)\) to match the value of the counter. To modify resources, Iris has a designated modality: the update modality \(\Rightarrow \). Intuitively, \(\Rightarrow P\) means \(P\) holds after (possibly) performing some updates to the resources.

Understanding the basics of the update modality is essential for understanding how later credits work, so here we take a closer look. First of all, each resource type (like the monotone counter) comes with resource-specific rules for updating its resources using the update modality. For example, the update rule for monotone counters is \(\Rightarrow \text{mono}_\gamma(n) \vdash \Rightarrow \text{mono}_\gamma(n + 1) \cdot \text{lb}_\gamma(n + 1)\) — i.e., we can update the ownership of \(\text{mono}_\gamma(n)\) to ownership of \(\text{mono}_\gamma(n + 1)\) and \(\text{lb}_\gamma(n + 1)\).

Besides the resource-specific rules, there are three important update rules: \text{UpdReturn}, \text{UpdBind}, and \text{UpdExec}. The first two rules essentially express that the update modality is a monad: we can always introduce an update with \text{UpdReturn} (the monad return), and we can compose two updates with \text{UpdBind} (the monad bind). In the rule \text{UpdBind}, the so-called magic wand \(P \rightsquigarrow Q\) may be understood as an implication which transfers ownership. In other words, if we own \(P \rightsquigarrow Q\) and are willing to give up ownership of \(P\), then we get back ownership of \(Q\). (Entailments \(P \vdash Q\) can always be converted into ownership of a wand \(P \rightsquigarrow Q\).) The third rule, \text{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 \(\text{mono}_\gamma(n)\) to \(\text{mono}_\gamma(n + 1)\)), 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 \(\Rightarrow P\) [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. To illustrate how that works in practice, we consider an example: verifying partial correctness of an infinite loop. To illustrate how that works in practice, we consider an example: verifying partial correctness of an infinite loop. To illustrate how that works in practice, we consider an example: verifying partial correctness of an infinite loop. To illustrate how that works in practice, we consider an example: verifying partial correctness of an infinite loop. To illustrate how that works in practice, we consider an example: verifying partial correctness of an infinite loop.

\[\Phi_{\text{loop}} \triangleq \{ \text{True}\} \text{loop } (\) \{v. \text{False}\}, \text{where loop } \triangleq \text{rec } x = \text{loop } x. \] (Of course, one can prove this result in logics without step-indexing; we focus on this toy example simply to avoid distractions that are orthogonal to showing how \(\Rightarrow \) works.)

To prove the triple \(\Phi_{\text{loop}}\), we need recursive reasoning. Unfortunately, because loop does not terminate, there is no inductive argument that we can use to prove \(\Phi_{\text{loop}}\). For such cases, step-indexed separation logics offer a solution based on step-indexing: Löb induction. Intuitively, Löb
induction can be understood as the coinduction principle of step-indexed separation logics. The Löb rule means 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, if we take a pure step (i.e., a step without non-determinism and state).

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

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$^2$ $[\mathcal{R}]^N$, 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 keeping 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 $\Phi_{\text{loop}}$). However, often, they do not align with program steps. Then we still have two options left: 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 $\triangleright (\exists n : N. \, \ell \mapsto n \, \triangleright \text{mono}_y (n))$. Using the commuting rules LaterExists and LaterSep, we can move the later inwards and obtain $(\exists n : N. \, \triangleright n \mapsto n \, \triangleright \text{mono}_y (n))$. Now, since $\ell \mapsto n$ is timeless (i.e., independent of Iris’s step-indexing mechanism), we can use the rules PointsToTimeless and Timeless 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 $\ell$.

In some cases, however, none of these techniques (i.e., eliminating later after steps, commuting rules, and timelessness) apply. We have already encountered (in §1) one example of this phenomenon: nested invariants. 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 $\ell n$, called the later credits, and a new update modality $\triangleright_{\text{le}} P$, called the later elimination update. Intuitively, one can think of owning $\ell n$ as the right to eliminate $n$ later modalities, and of the later elimination update $\triangleright_{\text{le}} P$ 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 steps:

1. We receive later credits by taking program steps. For example, we receive one later credit $\ell 1$ 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 symbolic execution rules for load, store, allocation, etc. similarly generate one credit after the step.)

$^2$As mentioned in the introduction, invariants carry namespaces $N$ and Hoare triples carry masks $E$ to avoid reentrancy (i.e., opening the same invariant twice at the same time). Additionally, both Iris’s update modality ($\triangleright$) and our new later elimination update modality ($\triangleright_{\text{le}}$) carry a mask because invariants can also be opened around them (see InvOpenUpd). Both namespaces and masks are orthogonal to later credits, but clutter the presentation. We therefore omit masks in all rules that are not concerned with invariants, in which case the reader can think of them as the set of all masks $\top$. 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 received credits, we can combine and split them freely with \texttt{CreditSplit}. With these rules, we have not eliminated any laters yet; that happens in the second step.

(2) We spend later credits through the later elimination update \(\Rightarrow_le\). That is, with \texttt{LEUpdLater} we can give up a credit \(\ell 1\), and, in exchange, eliminate a later modality by updating \(\Rightarrow P\) to \(P\).

In particular, we can use this rule to eliminate a guarding later from one of our assumptions. The update modality \(\Rightarrow le P\) can then be executed in various places in the logic. For example, just like with standard updates \(\Rightarrow P\), we can execute them in the precondition of Hoare triples with \texttt{LEUpdExec}.

Is all we did here replace one modality (\(\Rightarrow\)) with another one (\(\Rightarrow le\))? No, far from it. There are two key distinctions between the two modalities. The first one is that (\(\Rightarrow le\)) is a monad (analogous to \(\Rightarrow\)) with \texttt{LEUpdReturn} and \texttt{LEUpdBnd}, whereas (\(\Rightarrow\)) is not (it is only an applicative functor). As a monad, (\(\Rightarrow le\)) is considerably more pleasant to use than (\(\Rightarrow\)) since it is more compositional. For example, with \texttt{LEUpdBnd}, we can compose one later elimination update with another one. As a consequence, it is trivial to prove, for example, transitivity (\(i.e., \Rightarrow le P \Rightarrow le Q\)), or to use one later elimination update to spend two credits and eliminate two laters. In contrast, (\(\Rightarrow\)) does not satisfy the analogous rule \(\Rightarrow P = \Rightarrow P\), so we cannot use one later to eliminate two laters.

The second key distinction is that \(\Rightarrow le P\) can be executed virtually everywhere in the logic, whereas the elimination of laters is quite restricted (as we have explained above). The reason is that \(\Rightarrow le P\) generalizes the update modality \(\Rightarrow P\), which can be executed virtually everywhere. Thus, to integrate \(\Rightarrow le\) into Iris, we replace the update \(\Rightarrow P\) with \(\Rightarrow le P\) in most of Iris (explained in §5). This modification allows us to execute \(\Rightarrow le P\) where we could execute \(\Rightarrow P\) before.

\textbf{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. Similar to the verification of the monotone counter in §2.1, we do not need later credits here—we could use timeless and later commuting for this example as well. But it is nevertheless instructive as a baby example:

\[
\forall n. \{ n \in \mathbb{N} \} \Rightarrow f n \{ m. m \in \mathbb{N} \}
\]

We execute 41 + 1 with \texttt{PureStep} and thereby obtain a new later credit \(\ell 1\). We are left with proving \(\exists n : \mathbb{N}. l \mapsto n^N \vdash \{ \ell 1 \} l \leftarrow f(41 + 1) \{ v. \text{True} \}\). We frame the credit \(\ell 1\) around the call
of \( f \) with \texttt{FRAME}, leaving us to prove \( \exists n : \mathbb{N}, \ell \mapsto n \mid \{ \ell \} l \leftarrow m \{ v \text{ True} \} \) for some \( m \in \mathbb{N} \). After opening the invariant with \texttt{INVOPEN}, we have to show \( \{ 1 \mapsto (3n : \mathbb{N}, \ell \mapsto n) \} l \leftarrow m \{ v \mapsto (\exists n : \mathbb{N}, \ell \mapsto n) \} \). We spend the later credit to eliminate the later modality with \texttt{LEUPDLATER}, leaving us to prove \( \{ \Rightarrow \ell (3n : \mathbb{N}, \ell \mapsto n) \} l \leftarrow m \{ v \mapsto (3n : \mathbb{N}, \ell \mapsto n) \} \). Subsequently, we execute the later elimination update with \texttt{LEUPDEXEC}, leaving us to prove \( \{ 3n : \mathbb{N}, \ell \mapsto n \} l \leftarrow m \{ v \mapsto (3n : \mathbb{N}, \ell \mapsto n) \} \). The rest of the proof is routine, using \texttt{LATERINTRO} in the postcondition.

### 2.3 Prepaid Invariants with Later Credits

To see where later credits shine, we now consider a more serious example: prepaid invariants. Recall (from §1) that the unique advantage of prepaid invariants, in contrast to standard invariants, is that they can be opened \textit{without} a guarding later (see \texttt{INVPREOPEN}). To explain how that works and where later credits fit in, we first introduce a basic version \( R_{\text{basic}}^N \), which requires paying a later credit \( e \) 1 to close it, and then show how to get the full version \( R_{\text{pre}}^N \) satisfying \texttt{INVPREOPEN}.

\textbf{Basic prepaying.} In the example in §2.2, we have seen an example of prepaid reasoning. Specifically, we have seen how to obtain a credit \( e \) 1, keep it for several steps, and then spend it to eliminate the guarding later from the contents of an invariant. For our basic prepaid invariants, we use the same idea, but \textit{bundle a credit with an invariant}. Specifically, we define \( R_{\text{basic}}^N \triangleq R^N \cdot e \). As we explain below, we can always use the credit in the invariant to remove the guarding later from \( R \) when we open the invariant. We obtain two proof rules for basic prepaid invariants:

\[
\frac{R_{\text{basic}}^N}{P \cdot R_{\text{basic}}^N \in \{ v, Q \}} \quad \frac{R_{\text{basic}}^N}{P \cdot e \cdot R_{\text{basic}}^N \in \{ v, Q \}}
\]

To explain the use of later credits, we sketch the proof of \texttt{INVBASEOPEN}. Since \( e \) is atomic, we can open the invariant behind \( R_{\text{basic}}^N \) with \texttt{INVOPEN}. We have to prove \( \{ \Rightarrow (R \cdot e) \cdot P \} \in \{ v \mapsto (R \cdot e) \cdot Q \} \}

The later credits \( e \cdot n \) are timeless (see \texttt{CREDITTIMELESS}), and hence we can use \texttt{LATERSEP} and \texttt{TIMELESS} to eliminate the guarding later from \( e \cdot 1 \). Thus, we are left to prove \( \{ \Rightarrow (R \cdot e) \cdot P \} \in \{ v \mapsto (R \cdot e) \cdot Q \} \}

Next, we can use the later credit \( e \cdot 1 \) to eliminate the guarding later in front of \( R \) similar to the example in §2.2 (i.e., with \texttt{LEUPDLATER} and \texttt{LEUPDEXEC}). Thus, we are left with \( \{ R \cdot e \} \in \{ v \mapsto (R \cdot e) \cdot Q \} \}

We apply \texttt{LATERINTRO} in the postcondition, and are left with the goal \( \{ R \cdot e \} \in \{ v \mapsto (R \cdot e) \cdot Q \} \}

which is our assumption.

The basic prepaid invariants \( R_{\text{basic}}^N \) are already quite useful: when we open these invariants, we do not have to eliminate any later modalities, since \( R \) is not guarded by a later modality. In exchange for this liberty, we have to put back one credit \( e \cdot 1 \) after executing \( e \)—a credit which we typically obtain from the step of \( e \) (e.g., with \( \{ \ell \mapsto v \} ! \{ w, v = v \mapsto \ell \mapsto v \cdot e \} \}). Unfortunately, there is a limitation. If we want to open two invariants \( P_{\text{basic}}^N \) and \( Q_{\text{basic}}^N \) while reasoning about the same single physical step of executing \( e \), then we need to give back two credits, but the execution of \( e \) only generates one credit. So, next, we show how we extend the basic \( R_{\text{basic}}^N \) to the full \( R_{\text{pre}}^N \), which enables opening multiple (and nested) invariants.

\textbf{Time receipt extension.} The problem that prohibits opening multiple basic prepaid invariants is that we do not generate enough credits. Thus, to get prepaid invariants that satisfy \texttt{INVPREOPEN}, we need to increase the number of credits that are generated on each program step. Toward this end, we show how after we have reasoned about \( n \) program steps, we can subsequently generate \( n \) later credits \textit{per step}, by incorporating the notion of \textit{time receipts} (due to Mével et al. [2019]) into
our logic. The idea of time receipts is that with each program step, we obtain a time receipt $\mathbb{1}$—a witness for the program step. (They were used originally to prove lower bounds on time complexity.) Combined with later credits, this means that each execution step now produces both a later credit \(\text{and a time credit} \) (e.g., see PureStepReceipt). In addition, we can use the time receipts to get later credits with ReceiptCredits: if we own a receipt for \(n\) steps, then we can leverage it to generate \(n\) fresh later credits after the execution of \(e\).

With time receipts in hand, we define \(\mathcal{R}^N_{\text{pre}} \triangleq \mathcal{R} * \mathbb{1}^N\)\(\mathcal{L}\) and obtain the rules:

\[
\text{InvPreAlloc} \quad \begin{array}{c}
P \vdash \mathcal{R}^N_{\text{pre}} \mathcal{E} \vdash Q \quad e \{v, Q\} \\
\{P \vdash \mathbb{1}^N_\mathcal{L} \mathcal{E} \vdash R\} e \{v, Q\}
\end{array}
\]

\[
\text{InvPreOpen} \quad \begin{array}{c}
\mathcal{R} \vdash P \mathcal{E} \vdash Q \quad e \{v, Q\} \\
\{\mathcal{R}^N_{\text{pre}} \vdash P\} \mathcal{E} \vdash Q \quad e \text{ physically atomic}
\end{array}
\]

To understand how time receipts help us, we proceed with the proof of InvPreOpen. As in the proof of InvBasicOpen, we open the invariants behind \(\mathcal{R}^N_{\text{pre}}\) to obtain \(\#(R * \mathbb{1}^N)\). We then eliminate the later from \(\mathbb{1}\) and \(\mathbb{1}\), leveraging the fact that \(\mathbb{1}\) is, ironically, timeless (see ReceiptTimeless). Once again, we use the later credit \(\mathbb{1}\) to eliminate the later from \#(\(R\)), obtaining \(R\). Then, after applying LaterIntro in the postcondition, we are left with the goal \(\{R \vdash \mathbb{1}^N\} e \{v, R * \mathbb{1}^N\} \{e\} \mathcal{E} \mathcal{N}\). We use our receipt \(\mathbb{1}\) to generate a new credit with ReceiptCredits, and the resulting goal matches the premise of InvPreOpen.

With the time receipt extension of later credits, we have unlocked the full power of prepaid invariants. When we create one with InvPreAlloc, we have to give up one credit \(\mathbb{1}\) and one time receipt \(\mathbb{1}\). Afterwards, we can always use InvPreAlloc to open the invariants around Hoare triples without a later guarding the contents. Unlike InvBasicOpen, we can apply InvPreOpen multiple times if we want to open two (or more) invariants \(P_{\text{pre}}^N\) and \(Q_{\text{pre}}^N\). In particular, we can use InvPreOpen to open nested invariants (e.g., \(\exists \ell. \exists n : \mathbb{N}, \ell \mapsto n_{\text{pre}}^N \mathbb{L} \mapsto \gamma \mapsto \mathbb{L}^N_{\text{pre}} \mathcal{E} \mathcal{N}\) from \$1\) without any difficulty.

3 LATER CREDITS FOR REVERSE REFINEMENTS

For our next application of later credits, we show how they can be used to address a limitation with step-indexed logical relations described by Svendsen et al. [2016]. The issue arises when there is a function \(f : \tau \rightarrow \tau\) and we want to show for all \(e : \tau\) that \(f(e)\) is contextually equivalent to \(e\) at type \(\tau\), written \(f(e) \equiv_{\text{ctx}} e : \tau\). One strategy to show such an equivalence is to split it into proving two contextual refinements: we show \(f(e) \leq_{\text{ctx}} e : \tau\) and \(e \leq_{\text{ctx}} f(e) : \tau\) where \(\leq_{\text{ctx}}\) is contextual refinement. To prove these contextual refinements, we show that the expressions logically refine each other, according to a step-indexed logical relation. That is, we show \(f(e) \leq_{\text{log}} e : \tau\) and \(e \leq_{\text{log}} f(e) : \tau\), where \(\leq_{\text{log}}\) is a step-indexed judgment which (by the logical relation’s soundness theorem) implies \(\leq_{\text{ctx}}\).

Generally, when proving a logical refinement of the form \(e_1 \leq_{\text{log}} e_2 : \tau\), steps of \(e_1\) allow elimination of laters. Thus, showing \(f(e) \leq_{\text{log}} e : \tau\) is usually relatively straightforward, as far as step-indexing goes, since evaluating \(f(e)\) takes steps, which provides opportunities to eliminate laters. On the other hand, showing \(e \leq_{\text{log}} f(e) : \tau\), which we call the reverse refinement, can be problematic. Since we want to show \(e \leq_{\text{log}} f(e) : \tau\) for all \(e : \tau\), we cannot use steps taken by \(e\) to eliminate laters because \(e\) could be a value that does not take any steps at all.

\(\text{3The idea of incorporating time receipts is inspired by a recent extension of Iris [Jourdan 2021], where time receipts are used to support eliminating \(n\) laters immediately after the \(n\)-th program step. As we explain in \$7, the restriction to only eliminating laters right after program steps is not sufficient for our purposes, so we adapt it in this work (see \$3) to generate \(n\) later credits instead, and then derive new reasoning principles such as ReceiptCredits.}\)
checkCache $c\;x \triangleq \text{case } !c\text{ of }\text{none } \Rightarrow \text{none } | \text{some}(y, v) \Rightarrow \text{if } x = y \text{ then some}(v) \text{ else } \text{none}

\text{memo } f \triangleq \text{let } c = \text{ref}(\text{none});

\lambda x. \text{case } \text{checkCache } c\;x\text{ of }\text{some}(v) \Rightarrow v

| \text{none } \Rightarrow \text{let } v = f\;x; c \leftarrow \text{some}(x, v); v

Fig. 4. Implementation of a concurrent memoization function.

To address this issue with later elimination in the reverse refinement, Svendsen et al. [2016] use a transfinite step-indexed logical relation. Transfinite step-indexed models come with some drawbacks (discussed in §6), so in this work we show that later credits are an alternative solution for proving reverse refinements that avoids the need for transfinite step indexing.\textsuperscript{4} We demonstrate this by adding later credits to ReLoC [Frumin et al. 2018, 2021], a framework for proving contextual equivalences using step-indexed logical relations encoded in Iris. Using this new version of ReLoC we have verified the reverse refinement example considered by Svendsen et al. [2016], as well as a more complicated concurrent memoization example. We focus on the memoization example here.

The key idea is simple: rather than trying to prove $\vdash e \leq_{\log} f(e) : \tau$, we instead prove $\vdash \text{let } n = \Box n \vdash e \leq_{\log} f(e) : \tau$, where $n$ is some number of additional later credits and time receipts that are needed to establish the refinement. The logical relation’s soundness theorem is extended to say that $\vdash \text{let } n = \Box n \vdash e_1 \leq_{\log} e_2 : \tau$ implies $e_1 \leq_{\text{ctx}} e_2 : \tau$ for any $n$. Taking advantage of these additional credits and receipts requires changing the interpretation of types in the logical relation, as we will see. But before we get there, we describe our memoization example in more detail.

**Example: memoization of repeatable functions.** Memoization is a strategy to improve performance of an algorithm by caching computed values. Figure 4 gives a simple implementation of a concurrent memoization routine that caches the most recently computed value of a function. Given a function $f$ as input, memo $f$ first allocates a reference cell $c$ to store cached results. Then, it returns a new function that, when applied to an argument $x$, first checks the contents of $c$ to see if there is a cached value for the argument $x$, and returns it if so. If not, it evaluates $f\;x$ to get some value $v$, and then stores some $(x, v)$ in $c$.

Suppose $f : \tau_1 \rightarrow \tau_2$. When is memo $f \equiv_{\text{ctx}} f : \tau_1 \rightarrow \tau_2$? First, we need to require $\tau_1$ to be an equality type, meaning that values of type $\tau_1$ can be tested for equality (e.g. $\text{int}$, or $\tau_1 \times \tau_2$, where $\tau_1$ and $\tau_2$ are respectively equality types). Second, if $f$ has observable side-effects, then this equivalence may not hold, since the memoized version will run $f$ fewer times if there is a cache hit. However, if re-running ($f\;x$) always returns the same value and has no observable side-effects after the first run, then we should expect memo $f \equiv_{\text{ctx}} f : \tau_1 \rightarrow \tau_2$. We call such functions repeatable.

The existing type system for the language found in ReLoC, which is a version of System F extended with recursive types, mutable references, and concurrency features, is not rich enough to state that a function is repeatable in this sense. To be able to state a contextual equivalence formally, we thus extend the type system in ReLoC with a new type $\tau_1 \rightarrow^{\text{rep}} \tau_2$ of repeatable functions. The typing rule for this type uses a new typing relation, $\Gamma \vdash^{\text{rep}} e : \tau$, which implies that $e$ is a repeatable expression of type $\tau$. This judgment is a restriction of the standard typing judgment $\vdash$ that removes the rules for operations that have side effects.\textsuperscript{5} We then extend the standard typing judgement $\vdash$-

\textsuperscript{4}In §6, we will see how later credits and transfinite step-indexing are complementary approaches.

\textsuperscript{5}The fragment $\vdash^{\text{rep}}$ is somewhat limited since expressions may not contain instructions with side-effects. It is possible to bend this limitation. In the following, we develop a logical relation for $\vdash^{\text{rep}}$ which admits additional terms that cannot be typed
with two new rules for introducing and eliminating terms of type $\text{rep}$:

\[
\begin{align*}
\Gamma, x : \tau_1, f : \tau_2 \vdash e : \tau_2 & \quad \Gamma \vdash f : \tau_1 \rightarrow \text{rep} \tau_2 \\
\Gamma \vdash (\text{rec } f \ x = e) : \tau_1 & \rightarrow \text{rep} \tau_2 \\
\Gamma \vdash e : \tau_1 & \rightarrow \text{rep} \tau_2
\end{align*}
\]

With these definitions in place, we can formally state the desired result: for all $f$, if $f : \tau_1 \rightarrow \text{rep} \tau_2$ and $\tau_1$ is an equality type, then $\text{memo } f \equiv \text{ctx } f : \tau_1 \rightarrow \tau_2$. Proving this contextual equivalence runs into the issue with the reverse refinement described above: when trying to show $f \preceq_{\text{log}} \text{memo } f$, we have the problem that we need to eliminate lateres, but $f$ is an arbitrary function value that may not have any spare steps to take. Adding later credits to the interpretation of function types will allow us to work around this issue.

**Background: logical relations in Iris.** We first recall the basics of how the step-indexed logical relation in ReLoC is defined. ReLoC uses Iris’s program logic to specify the behavior of programs. This means we need 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 inside of Iris’s unary logic, ReLoC uses a technique from CaReSL [Turon et al. 2013], in which a second program is represented by ghost state in Iris (this technique has also been used in other formalization of 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]). This ghost state has assertions of the form $j \Rightarrow e$ which mean that thread $j$ in this ghost program is executing expression $e$. Similarly, there are ghost assertions of the form $\ell \mapsto_{\text{src }} v$ which mean that location $\ell$ points to $v$ in the ghost program’s state. The ghost program is “executed” by modifying the ghost state with the update modality. For example, to perform a store of $w$ to reference cell $\ell$ in the ghost program, we have the rule $j \Rightarrow (\ell \leftarrow w) * \ell \mapsto_{\text{src }} v \Rightarrow j \Rightarrow () * \ell \mapsto_{\text{src }} w$, which reflects that the store returns the unit value $()$, and the reference cell now contains the value $w$.

To prove a relational property about programs $e_1$ and $e_2$, it suffices to prove a Hoare triple about $e_1$ in which the precondition has a ghost thread running $e_2$ in an arbitrary evaluation context $K$. We define an assertion in Iris that expresses this relational pattern. Given $P : (\text{Val} \times \text{Val}) \rightarrow iProp$ where $iProp$ is the type of Iris assertions, we define

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

The adequacy theorem of Iris then ensures that, if $\vdash e_1 \preceq e_2 : P$, 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.

With this method of encoding relational properties, we can now define a logical relation. To simplify the explanation here, we leave out the details of how this approach scales to polymorphic and recursive types, since the addition of later credits that we describe later does not affect that part of the logical relation. The logical relation is defined in three steps:

(1) First, we define a type interpretation function $[-] : Type \rightarrow (\text{Val} \times \text{Val}) \rightarrow iProp$, That is, for each type $\tau$, $\llbracket \tau \rrbracket$ is an Iris relation on values.

(2) Next, this type interpretation is used to define a logical refinement relation on closed expressions $(e_1 \preceq_{\text{log}} e_2 : \tau) \triangleq (e_1 \leq e_2 : \llbracket \tau \rrbracket)$. This definition of logical refinement uses the above encoding of relational reasoning in Iris, with a postcondition that requires the values the expressions reduce to to be related according to $\llbracket \tau \rrbracket$.

syntactically in the side-effect free fragment $i_{\text{rep}}$, but are semantically repeatable. For example, $\lambda()$. let $r = \text{ref} (41) ; !r + 1$ is semantically repeatable even though it has side effects from the perspective of $i_{\text{rep}}$. 
\[\int \triangleq \lambda (v_1, v_2), \exists z \in \mathbb{Z}. v_1 = v_2 = z\]
\[\tau \rightarrow \tau' \triangleq \lambda (v_1, v_2). \forall u_1, u_2. \Box ([\tau](u_1, u_2) \rightarrow (v_1 u_1) \leq (v_2 u_2) : [\tau'])\]
\[\text{ref } \tau \triangleq \lambda (v_1, v_2). \exists \ell_1, \ell_2. v_1 = \ell_1 * v_2 = \ell_2 * \exists u_1, u_2. \ell_1 \mapsto u_1 * \ell_2 \mapsto_{\text{ref}} u_2 * [\tau](u_1, u_2)N_{\ell_1, \ell_2}\]

Fig. 5. Type interpretation \(\llbracket - \rrbracket\) in ReLoC. (Polymorphic types, pairs, sums, and recursive types omitted.)

(3) Finally, the logical refinement relation is lifted from closed expressions to open expressions. Given a type context \(\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n\) and open terms \(e, e'\), we define:

\[\Gamma \vdash e \leq_{\log} e' : \tau \triangleq \forall v_1, v_1', \ldots, v_n, v_n'. \bigstar_{i=1,\ldots,n} [\tau_i](v_i, v_i') \triangleright e[v_1/x_1] \cdots [v_n/x_n] \leq_{\log} e'[v_1/x_1] \cdots [v_n'/x_n] : \tau\]

Figure 5 gives an excerpt of the definition of \(\llbracket - \rrbracket\) in ReLoC. The two most interesting cases are for \(\tau \rightarrow \tau'\) and \(\text{ref } \tau\). The former says that two values are related at type \(\tau \rightarrow \tau'\), if whenever they are applied to values that are assumed to be related at type \(\tau\), the resulting application expressions are related at the interpretation of \(\llbracket \tau' \rrbracket\). In the definition of \(\llbracket \tau \rightarrow \tau' \rrbracket\), we use Iris’s persistence modality \(\Box\ P\), which ensures that the assertion \(P\) does not own any non-duplicable resources. We will expand on the persistence modality and its use shortly. For \(\text{ref } \tau\), 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 \(\tau\). (To keep the changes to ReLoC specific to reverse refinements, we do not use a prepaid invariant here.) 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 when trying to define 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 \(\Gamma \vdash e_1 \leq_{\log} e_2 : \tau\) then \(\Gamma \vdash e_1 \leq_{\text{ctxt}} e_2 : \tau\)

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

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. This theorem is proved by showing that the logical relation is a congruence relation wrt. all typing rules.

Because the type system here is not sub-structural, a key component of this proof is that the \(\llbracket - \rrbracket\) predicate is duplicable for all types. This means that when trying to prove \(\Gamma \vdash e_1 \leq_{\log} e_2 : \tau\), we can duplicate the assumptions about the values substituted in for the variables in \(\Gamma\). This duplicability requirement is what forces us to include the \(\Box\) modality in the definition of \(\llbracket \tau \rightarrow \tau' \rrbracket\) above. The modality \(\Box P\) requires us to prove \(P\) without any assertions that we own exclusively (e.g., \(\ell \mapsto v\)). We may only use duplicable assertions (e.g., \(\llbracket P \rrbracket^N\)) and, as a result, \(\Box P\) is also duplicable.

**Encoding repeatability as an Iris assertion.** To extend this logical relation to support the repeatability type judgment \((\rightarrow^{\text{rep}})\) and repeatable function type \((\rightarrow^{\text{rep}})\), we need an Iris assertion that captures that an expression is repeatable. The impredicative features of Iris make this relatively straightforward. First, for repeatability of ghost state programs, we define:

\[
\text{repGhost}(e, v) \triangleq \Box(\forall j, K. j \implies K[e] \rightarrow \Box j \implies K[v])
\]
That is, \( \text{repGhost}(e, \nu) \) says that for any ghost thread running \( e \) in an evaluation context \( K \), we can perform a ghost update to "execute" \( e \) to the value \( \nu \). The persistence modality \( \Box \) ensures that this can be done as often as we like. For non-ghost code, we have

\[
\text{replImpl}(e, \nu) \triangleq \{ \text{True} \} e \{ \nu' : \nu = \nu' \}
\]

Just as with \( \text{repGhost} \), if \( \text{replImpl}(e, \nu) \) holds, then in the course of a proof we can "run" \( e \) and it can only terminate in value \( \nu \). We do not need \( \Box \) here, because it is baked into Hoare triples (see §5.2).

Using these definitions, we define a repeatable form of \( e_1 \leq e_2 : P \) as follows:

\[
( e_1 \leq^{\text{rep}} e_2 : P ) \triangleq ( e_1 \leq e_2 : \lambda(v_1, v_2).P(v_1, v_2) \ast \text{replImpl}(e_1, v_1) \ast \text{repGhost}(e_2, v_2) )
\]

This requires that the results of evaluating \( e_1 \) and \( e_2 \) not only need to be related according to \( P \), but also we must have proofs that the expressions can repeatably run to those values. Repeatable versions of the key definitions in the logical relation are then obtained by using \( \leq^{\text{rep}} \) in place of \( \leq^{\text{log}} \):

\[
\begin{align*}
\llbracket \tau \rightarrow^{\text{rep}} \tau' \rrbracket & \triangleq \lambda v_1, v_2. \forall u_1, u_2. \Box(\llbracket \tau \rrbracket(u_1, u_2) \rightarrow (v_1, u_1) \leq^{\text{rep}} (v_2, u_2) : \llbracket \tau' \rrbracket) \\
(e_1 \leq^{\text{rep}} e_2 : \tau) & \triangleq (e_1 \leq^{\text{rep}} e_2 : \llbracket \tau \rrbracket)
\end{align*}
\]

Finally, \( \leq^{\text{rep}} \) is lifted to open terms in the same way as \( \leq^{\text{log}} \) above. In addition to soundness, the resulting logical relation has an extended form of the fundamental property, as follows:

**Theorem 3.3 (Fundamental Property).**

1. If \( \Gamma \vdash e : \tau \) then \( \Gamma \models e \leq^{\text{log}} \nu : \tau \).
2. If \( \Gamma \vdash e : \tau \) then \( \Gamma \models e \leq^{\text{rep}} \nu : \tau \).

**The need for later credits.** Despite the addition of repeatability to the logical relation, the definition of the logical relation is still problematic if we try to prove \( f \leq^{\text{ctx}} \text{memo} f : \tau_1 \rightarrow \tau_2 \), the reverse refinement in our memoization example. Let us see where the issue is. We assume that we have a value \( f : \tau_1 \rightarrow^{\text{rep}} \tau_2 \), where \( \tau_1 \) is an equality type and we want to show that \( f \leq^{\text{log}} \text{memo} f : \tau_1 \rightarrow \tau_2 \). By the fundamental lemma, we know that \( f \leq^{\text{log}} \nu \rightarrow^{\text{rep}} \tau_2 \).

Proceeding with the proof by unfolding the definitions and introducing universally quantified variables, we need to prove \( \{ j \Rightarrow K[\text{memo} f] \} f \{ \nu, \exists \nu'. j \Rightarrow K[\nu'] \ast \llbracket \tau_1 \rightarrow \tau_2 \rrbracket(\nu, \nu') \} \).

By performing updates to evaluate memo \( f \), we first allocate some ghost reference cell \( c \) for storing cached values and obtain \( c \leftarrow^{\text{src}} \text{none} \). We are left with a value \( f' \), where

\[
f' = \lambda x. \text{case} \text{checkCache} c x \text{ of} \text{some}(\nu) \Rightarrow \nu
\]

and we must prove \( \llbracket \tau_1 \rightarrow \tau_2 \rrbracket(f, f') \). Our proof of this must be duplicable (due to \( \Box \)), yet we will need to use the points-to fact for \( c \) when reasoning about the execution of \( f' \), and points-to facts are not duplicable. To resolve this issue, we create the following invariant for the points-to fact of \( c \):

\[
\begin{align*}
( & \exists y, v, v'. c \leftarrow^{\text{src}} \text{some}(y, v') \ast \text{replImpl}(f y, v) \ast \text{repGhost}(f y, v') \ast \llbracket \tau_1 \rrbracket(y, y) \ast \llbracket \tau_2 \rrbracket(\nu, \nu') ) \\
\lor & (c \leftarrow^{\text{src}} \text{none} )
\end{align*}
\]

This invariant requires that if \( c \) contains a cached value, meaning \( c \leftarrow^{\text{src}} \text{some}(y, v') \) for some \( y \) and \( v' \), then there is some \( \nu \) such that \( f y \) repeatably runs to \( \nu \) on the implementation level and \( v' \) on the ghost level, where \( \nu \) and \( v' \) are related according to the interpretation of \( \tau_2 \).

Assuming this invariant, we must now show that for any arguments \( x, x' \) such that \( \llbracket \tau_1 \rrbracket(x, x') \), we have \( f x \leq f' x' : \llbracket \tau_2 \rrbracket \). Here, we use a lemma that tells us that since \( \tau_1 \) is an equality type, \( \llbracket \tau_1 \rrbracket(x, x') \) implies \( x = x' \). Thus, unfolding the definitions, that means we must show

\[
\{ j \Rightarrow K[f' x]\} f x \{ \nu, \exists \nu'. j \Rightarrow K[\nu'] \ast \llbracket \tau_2 \rrbracket(\nu, \nu') \}
\]
To execute ghost steps of \( f' x \) we need access to the points-to assertion for \( c \) in the invariant. To do so, we use a feature of Iris’s invariants that we have largely ignored so far: with \( \text{InvOpenUpd} \), we can open them (guarded by a later), perform some ghost updates, and then close them again—without taking a step. (From a concurrency perspective, using the invariant for zero steps is fine, because no other thread can observe it in between.)

In our case, once we open the invariant (and use suitable commuting rules), we have to consider two cases. If we are in the “\( c \mapsto_{\text{src}} \text{some} \)” branch, there is no issue: both the real code and the ghost code will end up executing \( f x \). We can update the source to advance to the execution of \( f x \) and then execute \( f x \) in both ghost code and implementation. From the definition of \( [\tau_1 \rightarrow^\text{rep} \tau_2] \), after this code executes, we get \( \text{replImpl}(f x, v) \) and \( \text{repGhost}(f x, v') \) for the returned values, which we store in the invariant (again opening it for zero steps).

For the “\( c \mapsto_{\text{src}} \text{some}(y, v') \)” branch, we obtain:

\[
\exists y, v'. c \mapsto_{\text{src}} \text{some}(y, v') \implies \text{replImpl}(f y, v) \implies \text{repGhost}(f y, v')
\]

after applying commuting rules and using timelessness of \( c \mapsto_{\text{src}} \text{some}(y, v') \). If the cached argument \( y \) is not equal to \( x \), the argument is similar to the “\( c \mapsto_{\text{src}} \text{none} \)” case. The difficult case is when there is a cache hit (i.e., \( x = y \)). Then, we use the \( c \mapsto_{\text{src}} \text{some}(x, v') \) to execute the steps of \( f' x \) in the ghost program, which will return the cached value to the point where we own \( j \mapsto K[v'] \). Turning to reasoning about \( f x \) in the implementation (i.e., the Hoare triple), we have \( \triangleright \text{replImpl}(f x, v) \) from the invariant. Now, we would like to use this to argue that \( f x \) must return \( v \). But \( \text{replImpl}(f x, v) \) is not timeless (it is a Hoare triple), so we cannot eliminate the \( \triangleright \) and our proof attempt is stuck.⁶

### Adding later credits to the logical relation

We would like to have a later credit to eliminate the later guarding \( \text{replImpl}(f x, v) \) in the above proof. To do so, we will change the definition of \( [\tau_1 \rightarrow \tau_2] \) and \( [\tau_1 \rightarrow^\text{rep} \tau_2] \) to allow functions to demand later credits. This change creates a tension: since functions can now demand credits, to prove the fundamental lemma for the new definition, we need to be able to provide them whenever the function is applied to an argument. To resolve this tension we define receipt pools. A receipt pool, written \( \text{pool}(n) \), where \( n \) is a natural number, consists of \( n \) different invariants, each containing a time receipt \( \bullet 1 \). This assertion supports the following rules:

\[
\begin{align*}
\text{PoolExtend} & \quad & \text{PoolCredits} \\
\begin{array}{l}
\bullet 1 \ast \text{pool}(n) \vdash & \text{pool}(n + 1)
\end{array} & \quad & \begin{array}{l}
\{ P \ast \mathcal{E}(n + 1) \} e_1 \{ v. Q \} \quad & e_1 \rightarrow_{\text{pure}} e_2
\end{array}
\end{align*}
\]

That is, given a time receipt, we can extend an existing pool by 1. And, if we have a \( \text{pool}(n) \) we can generate \( n + 1 \) later credits after taking a step.⁷ A receipt pool is duplicable because it consists of multiple invariants, each of which is duplicable.

We then redefine the interpretation of \( \rightarrow \) to use receipt pools and later credits:

\[
\begin{align*}
\llbracket \tau \rightarrow \tau' \rrbracket \triangleq & \lambda v_1, v_2, \exists n, x_1, f_1, e_1, x_2, f_2, e_2. v_1 = (\text{rec } f_1 x_1 = e_1) \ast v_2 = (\text{rec } f_2 x_2 = e_2) \ast \text{pool}(n) \\
& \ast \forall u_1, u_2. \square (\llbracket \tau \rrbracket(u_1, u_2) \rightarrow \mathcal{E}(n + 1) \rightarrow (e_1[u_1/x_1] [v_1/f_1]) \leq (e_2[u_2/x_2] [v_2/f_2]) : \llbracket \tau' \rrbracket)
\end{align*}
\]

Let us break this definition down into pieces. First, we assert \( v_1 \) and \( v_2 \) are in fact functions. Then, for some existentially quantified \( n \), there must be a \( \text{pool}(n) \). Finally, given values \( u_1 \) and \( u_2 \) related according to \( \llbracket \tau \rrbracket \), as well as \( \mathcal{E}(n + 1) \), we substitute \( u_1 \) and \( u_2 \), and the recursive definitions of the functions into the function bodies. The resulting expressions must be related according to \( \llbracket \tau' \rrbracket \).

That is, whereas the earlier interpretation of \( \rightarrow \) was stated in terms of the applications of \( v_1 u_1 \) and

---

⁶We might try to take a step (by doing a beta reduction of \( f x \)) to remove the later, but then we would have to prove a triple about the expression that arises from taking the step instead of \( f x \), so we would no longer be able to use \( \text{replImpl}(f x, v) \).

⁷This rule relies on a modification of \text{ReceiptCredits}. 

---
we get £

The fundamental property holds with this new definition. The biggest change in proving the fundamental property is the case for the function application typing rule. In that case, given $\square \tau \rightarrow \tau'$, we must prove that $\nu_1 \nu_1 \leq \nu_2 \nu_2 : \square \tau'$. (This case was trivial with the original definition of $\square \tau \rightarrow \tau'$, since the desired conclusion was precisely the definition.)

Under the new definition, we have pool($n$) for some $n$ and need £($n+1$) after reducing the application by 1 step. We obtain this £($n+1$) by using the rule PoolCredits for the $\beta$-reduction step, and the rest of the case is straightforward.

In addition, we obtain a stronger version of the soundness theorem, where we may now assume £$n * \boxtimes n$ when proving two expressions are logically related:

**Theorem 3.4 (Soundness).** If (£$n * \boxtimes n \vdash \Gamma \models e_1 \equiv_\log e_2 : \tau$) then $\Gamma \vdash e_1 \equiv_{\text{ctx}} e_2 : \tau$

**Proving the memoization reverse refinement with later credits.** Returning to our memoization example, we use this extended soundness theorem to be able to start with an initial later credit and time receipt. That is, assuming $f : \tau_1 \rightarrow \text{rep} \tau_2$, where $\tau_1$ is an equality type, we prove £$1 + \boxtimes 1 \vdash f \equiv_\log \text{memo} f \vdash \tau_1 \rightarrow \tau_2$. As before, we apply the fundamental lemma to our assumption about $f$ to get that it is logically related to itself. From the new definition of $\square \tau_1 \rightarrow \text{rep} \tau_2$, we know that $f = (\text{rec} f_0 z = e)$ for some $z, f_0, e$, that there is pool($n$) for some existentially quantified $n$, and that $f$ demands £($n+1$) to be executed. Using PoolExtend, we exchange our $\boxtimes 1$ for pool($n+1$).

As before, we execute the ghost code in memo $f$ to allocate the reference cell $c$ and obtain $f'$. Our invariant storing the points-to for $c$ is slightly different, storing repeatability facts about the result of substituting values into the body $e$ of $f$:

$$
(\exists y, v, v'. c \mapsto_{\text{src}} \text{some}(y, v') * \text{replmpl}(e[y/z][f/f_0], v) * \text{repGhost}(e[y/z][f/f_0], v'))^N
$$

Now, when proving $\square \tau_1 \rightarrow \tau_2 (f, f')$ we choose the existentially quantified natural number in the interpretation of the arrow type to be $n+1$, since we have pool($n+1$). This choice means we get £($n+2$) now when reasoning about $f$ and $f'$ applied to some arguments that have been substituted in. The non-cache hit cases proceed as before, except that we have to give up £($n+1$) of our £($n+2$) to use our assumptions about $f$. In the case of a cache hit, we use £$1$ to remove the later from replmpl. This replmpl lets us show that the non-memoized code will return the value that is cached in the ghost code version, and thus complete the proof.

### 4 LATER CREDITS FOR LOGICAL ATOMICITY

In this section, we will explore how later credits help to alleviate a fundamental limitation of one of the primary uses of Iris: proving logical atomicity [Jung et al. 2015, 2020]. Derived from atomic specifications in TaDA [da Rocha Pinto et al. 2014], logical atomicity is the canonical way to express functional correctness of concurrent data structures in Iris: akin to linearizability, a logically atomic specification of an operation says that the operation appears to be executed atomically. Logical atomicity decouples the abstract atomic action from when it happens physically, which can cause clients a headache when they need to eliminate a later but cannot take any program steps. Later credits help us here—we can use them to eliminate laterss without any program steps being around.

In the following, we look at a particularly challenging logical atomicity proof—a concurrent counter with a backup, a data structure representative of algorithms that perform what we dubbed *unsolicited helping*. We explain unsolicited helping with the concurrent counter (in §4.1), its logically atomic specification (in §4.2), and how later credits enable its logical atomicity proof (in §4.3).
new() ≜ let (b, p) := (ref(0), ref(0)); // backup and primary
  fork {backup_thread(b, p)} ; (b, p)
backup_thread(b, p) ≜ b ← !p; backup_thread(b, p) // copy primary to backup, in a loop
await_backup(b, n) ≜ if !b < n then await_backup(b, n) else () // loop until !b reaches n
incr(b, p) ≜ let n = FAA(p, 1); // atomic increment (via fetch-and-add)
  await_backup(b, n + 1); n // return old (pre-increment) value
get(b, p) ≜ let n = !p; await_backup(b, n); n
get_backup(b, p) ≜ !b

Fig. 6. Counter with a backup

4.1 Unsolicited Helping: Counter with a Backup

The counter with a backup (shown in Figure 6) is basically a regular monotone counter with methods incr to increment the value of the counter by 1 and get to get the current value of the counter. There’s just one twist: the value of the counter is actually stored in two locations, the “primary” p and the “backup” b, and there is a second read operation get_backup that reads the current value of the counter from the backup instead of the primary. What makes this data structure interesting is that incr does not immediately update both primary and backup; instead, it updates the primary and waits for a background thread (here backup_thread) to synchronize that value with the backup. The primary and backup can hence get out-of-sync, and yet clients will not be able to distinguish get and get_backup.

This example may seem contrived, and indeed it is—but it captures the essence of 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 new current state of the key-value server (as observable by clients) will be exactly what was stored on disk at the time of the crash.

To avoid all the complexities of crashes and durable state, we have condensed this realistic 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 because there might be a crash at any time.

The point of the counter is to serve as an example for a data structure whose correctness we can prove with later credits, but that we do not know how to prove without. What makes the counter so challenging is that it does what we call unsolicited helping. In the context of linearizability, helping refers to data structures where one thread “helps” another to complete its desired action—the linearization point of an action run by one thread is actually located in code executed by another thread. In our example, this arises for get (and incr): the linearization point of get is at the read of p only if the backup has caught up with the primary already. Otherwise, the get is linearized the moment the backup thread catches up and bumps b to the value read by get (or a higher value). This means that the linearization point of get is actually inside backup_thread. Moreover—and this is where helping becomes “unsolicited”—the backup thread has no idea if there even is a get running concurrently that needs any helping, since the get do not explicitly “register” for being
helped. They are just "going along for the ride" when the backup thread is bumping the counter to help incr.

4.2 Logically Atomic Specification

What does it mean for a concurrent counter to be “correct”? We have seen a possible specification in §2.1, but this specification was rather weak: it does not exactly pin down the value returned by get, only providing a lower bound instead. We could instead interpret the assertion \( \text{counter}(c, n) \) as stating exact knowledge of the current value of the counter, but then we have to make this assertion non-duplicable ownership much like a points-to assertion. Such a specification would still be too weak: if every counter operation requires ownership of the counter, it is impossible to call multiple counter operations concurrently from different threads! No good for a concurrent counter.

So what can we do? We were on the right track with \( \text{counter}(c, n) \) representing ownership of the counter, we just need one more feature: we need to be able to open invariants around counter operations. After all, heap locations are exclusively owned by \( \ell \mapsto v \), and yet we can perform concurrent operations on the same location—by putting the points-to into an invariant, and opening that invariant around the operations! The trouble is that all the invariant opening rules we have seen so far have the side-condition that we can only open invariants around physically atomic operations. We thus need a way to express that counter operations can have invariants opened around them despite them not being physically atomic.

Enter logically atomic specifications. These triples written with angle brackets instead of curly braces satisfy the following proof rule:

\[
\begin{align*}
\text{LAInvOpen} & : \langle \forall R : \mathcal{E} \rangle e (v \mapsto R \ast P) \quad (v \mapsto R \ast Q) \subseteq \mathcal{E} \\
& \quad \frac{\langle R \rangle^{N \ast P} e (v \mapsto Q) \subseteq \mathcal{E} \setminus N}{\langle \mathcal{E} \rangle^{N \ast P} e (v \mapsto Q) \subseteq \mathcal{E} \setminus N}
\end{align*}
\]

This rule is sound because a logically atomic triple expresses that the implementation must atomically transition from the precondition to the postcondition. (The masks in logically atomic triples are inverted compared to regular Hoare triples, which is why the mask grows instead of shrinking when applying the rule to a goal.)

The logically atomic specification for the counter is shown in Figure 7. The specification is given in terms of two abstract predicates: \( \text{is\_counter}_{\mathcal{N}}^\gamma(c) \) says that the program value \( c \) represents a counter that needs namespace \( \mathcal{N} \) for its invariants (this is pure knowledge of the existence of that counter, without knowing its current value), and \( \text{counter}_{\gamma}(n) \) represents exclusive ownership of the current value of the counter. The logical name \( \gamma \) ties the two together. All operations take counter as their precondition and return counter with a possibly changed counter value in their postcondition. Other than that, the most remarkable thing to say about this specification is that it is not particularly remarkable: it looks exactly like how we would specify a sequential counter with regular Hoare triples, except that the brackets are more pointy. Just like how linearizability gives a
canonical correctness condition for the concurrent pendant to a sequential data structure; logical atomicity is the canonical concurrent pendant to sequential specifications, i.e., regular Hoare triples. There is just one syntactic difference: the current value \( n \) of the counter right before the operation is bound via a special binder that is embedded in the logically atomic triple. The reason is that this value can actually change during the execution of the operation because other threads might be calling \( \text{incr} \) concurrently. This binder represents values that are only determined when the operation actually happens (i.e., at the linearization point). If we would bind the \( n \) on the outside like in a regular specification, the client would have to fix \( n \) in advance, which would exclude mutations by other threads—the specification would be too weak.

**Logically atomic triples.** We have seen how to specify a concurrent counter with logically atomic triples, but how are these triples themselves defined? Atomic triples are defined in terms of regular Hoare triples, wrapping the atomic transition from precondition to postcondition in an atomic update \( \text{AU}(x. P, v. Q)^E_R : (x. P) \ell e (v. Q)^E_R \triangleq \forall R. \{ \text{AU}(x. P, v. Q)^E_R \} e (v. R) \)

For reasons of space, we cannot explain how exactly \( \text{AU}(x. P, v. Q)^E_R \) is defined. What it lets us do, however, can be summarized well: at the linearization point, we can “open” the atomic update to obtain the atomic precondition \( \exists x. P \) (i.e., we also learn the value for the special binder \( x \) at this point). This removes \( E \) from our mask. Within the same atomic step (logically or physically atomic), we have to give the atomic postcondition \( Q \) back in order to “close” the atomic update (and get back to our original mask). In so doing, we obtain the result \( R \) of the atomic update. Since \( R \) is also the postcondition of the Hoare triple we are proving for \( e \), we have to perform this interaction with the atomic update at some point during the proof, or else we are not going to be able to complete it.

### 4.3 Logical Atomicity with Later Credits

Let us now turn to the verification of the counter with a backup, the unsolicited helping example. The point of the counter is to serve as an example for a specification that we can prove with later credits, but that we do not know how to prove without. Unfortunately, the proof involves a significant amount of detail. To avoid losing the reader in the midst of its detail (which can be found in the supplementary material [Author(s) 2021]), we give a high-level explanation here what happens and where later troubles arise. There are two challenges that this proof poses:

1. As already explained, the algorithm relies on helping. Helping makes logical atomicity challenging to prove because the operation that is being helped needs to transfer its atomic update (e.g., \( \text{AU}(n. \text{counter}, r(n), m. m = n + \text{counter}, r(n))^E_R \) for get) to another thread, have that thread run the update, and then transfer the result \( (R) \) back. Such ownership transfer is arranged via invariants—and atomic updates are not timeless!

2. We use logical atomicity all the way down: to remain maximally modular (as it was originally intended [Jung et al. 2015]), we verify the logically atomic specification of the counter on top of a logically atomic heap. As a consequence, we have to execute, for example, the atomic update of \( \text{incr} \) \( \text{AU}(n. \text{counter}, r(n), m. n = m + \text{counter}, r(n + 1))^E_R \) as part of the atomic update of the store operation in the background thread (i.e., \( \text{AU}(v. \ell \mapsto v, \_ \ell \mapsto w)^E_R \)).

The second point means that we have to open the invariant used for helping while doing an atomic update, which is only possible via \text{InvOpenUpd}. The first point means that we cannot eliminate the later modality that \text{InvOpenUpd} adds in front of the atomic update!

Previous versions of Iris resolved this conundrum by exploiting that after a thread “knows that it should help someone” and before it actually does the helping, there are some physical steps happening—for example, after the backup thread read the primary and before it writes the new
value to the backup. This is sufficient for incr, because when we read the primary we can determine exactly which incr operations we are going to help by updating the backup. But for get, we are out of luck: new get calls waiting to be helped can appear any time up until the very moment that the backup value is actually bumped—the lack of physical communication between get and the backup thread (the entirely unsolicited nature of this helping) means that there is no physical step in the backup thread that we could use to eliminate a later.

And that’s why later credits are crucial here. We can simply arrange for each waiting get to put a credit into the invariant that manages the counter (get performs enough physical steps, so it has credits to spare), and then the backup thread can use those credits to eliminate the guarding later from the atomic update, and proceed as before. In fact, this works well for all kinds of helping: whoever needs help with linearization can “pay” for that helping by putting in a credit, thus removing the pressure from the helper to have to deal with all those later modalities. This even lets us entirely remove the clever “laterable” hack that Iris previously required to make regular helping work [Jung 2019]. Later credits solve the same problem in a more elegant way, and unlike the previous approach they can also handle unsolicited helping. We ported all major logical atomicity case studies to our new, simpler definition of logical atomicity to ensure that later credits can indeed entirely replace the previous, incomplete approach to tackling this problem.

5 MODEL AND SOUNDNESS

In this section, we explain how the later credit mechanism is implemented in the Iris framework. Iris (without later credits) factors into two layers: a base logic and a program logic. The base logic, in short, is a step-indexed logic of bunched implications [O’Hearn and Pym 1999]. What that means is that (1) the base logic is step-indexed (e.g., it has the later modality ▶ P), (2) it has the distinguishing connectives of separation logic (e.g., P * Q and P → Q), and (3) it has support for ghost state through updates (without masks) and resources. Importantly, the base logic does not yet have any notion of programs, state, or Hoare triples. These notions are defined in the program logic, which is built on top of the base logic (for a discussion of the definition see [Jung et al. 2018b]). The program logic defines (in terms of the base logic) a number of connectives that we have already encountered: updates with masks, invariants, Hoare triples, and logically atomic triples.

We insert later credits as an additional layer between the base logic and the program logic. We define later credits in terms of the base logic without changing its model, because, as it turns out, the base logic is expressive enough to define and prove adequacy of the later credits mechanism (see §5.1). Equipped with the later credits mechanism, we then redefine the program logic of Iris. For the most part, this change involves replacing standard updates with later elimination updates. The non-trivial change to the program logic is the way we alter Hoare triples and the proof that they entail partial correctness (in §5.2).

5.1 Later Credits and Later Elimination Updates

As we have already seen, the central device enabling the new reasoning principles presented in this work is the later elimination update. Below, we will introduce its definition and show how the interaction with later credits ensures its soundness.

Eliminating laters. As we have seen, the central feature of later elimination updates is that they allow us to eliminate laters. Thus, as a first approximation, suppose we define ▷ P ≜ (▷ P ∨ ▶ P). We can use ▷ P to eliminate a later (i.e., we have ▶ P ⊢ ▷ P) and to execute a standard update.

8To be precise, Iris has two kinds of update modalities: a basic update, which lives in the base logic and a fancy update ▷ P, which can interact with invariants and lives in the program logic. The latter is defined in terms of the former. We show how to integrate later credits into the basic update, which directly translates to a version of the fancy update with later credits.
(i.e., we have \( \models P \vdash \models^v_\text{le} P \)), and hence prove \textsc{LEupdReturn}. But unfortunately, this definition is not transitive: we do not have \( \models^v_\text{le} \models^v_\text{le} P \vdash \models^v_\text{le} P \), a direct consequence of the composition rule \textsc{LEupdBind}. For transitivity to hold with this definition, among others, the later modality would need to be transitive (i.e., \( \triangleright P \vdash \triangleright P \)), which it is not.

**Enabling transitivity.** Let us try to bake transitivity into the definition with recursion:\(^9\) \( \models^v_\text{le} P \triangleleft \models^v_\text{le} P \). This definition allows us to execute an update, and pick one of two cases: the base case or the recursive case. In the base case, we simply have to prove \( P \), and hence we have \( \models P \vdash \models^v_\text{le} P \). The recursive case is more interesting. We can use it to eliminate a later modality (i.e., we have \( \triangleright P \vdash \models^v_\text{le} P \)), and afterwards prove a later elimination update again.

Unfortunately, the approximation \( \models^v_\text{le} P \) is meaningless—for any proposition \( Q \), we have \( \models^v_\text{le} Q \). The reason is Löb induction. Suppose we want to prove \( \text{True} \vdash \models^v_\text{le} Q \). Then, by Löb, it suffices to prove \( \triangleright \models^v_\text{le} Q \vdash \models^v_\text{le} Q \), which follows from \( \triangleright P \vdash \models^v_\text{le} P \) (with \( P \triangleleft \models^v_\text{le} Q \)) and transitivity.

**Bounding laters.** Intuitively, the problem with \( \models^v_\text{le} P \) is that it allows us to eliminate an unbounded number of laters—thus enabling the inductive proof with Löb above. Here is where later credits come in: laters have a cost, and to eliminate them, we need to be ready to pay that cost, with later credits \( \ell n \).

We have already seen a part of the rules governing the resource \( \ell n \) (see Figure 3). To explain the definition of \( \models^v_\text{le} P \), there is however still one important puzzle piece missing, which ensures that we can only eliminate a bounded number of laters: the credit supply \( \ell n \). The credit supply \( \ell n \) is another resource, which interacts with the credits \( \ell n \) that are around. Specifically, the relationship is that at any point the credit supply \( \ell n \) is the sum of all the credits \( \ell n \).\(^{10}\) (Recall that we can split and combine credits with \textsc{CreditSplit}.) The relationship is manifested by:

\[
\begin{align*}
\textsc{SupplySup} & : \ell n \triangleright \ell m \vdash m \geq n \\
\textsc{SupplyDecr} & : \ell n \triangleright \ell (n + m) \vdash \ell m
\end{align*}
\]

The rule \textsc{SupplySup} guarantees that the credit supply \( \ell n \) is always an upper bound for the credits \( \ell n \) that are distributed in the logic (e.g., stored in invariants). The rule \textsc{SupplyDecr} allows us to decrease the credit supply \( \ell n (n + m) \) by the amount of later credits \( \ell n \) that we own.

**Later elimination update.** With the credit supply in hand, we are now ready to define the full version of the later elimination update:

\[
\models^v_\text{le} P \triangleleft \forall n. \ell n \vdash \models^v_\text{le} ((\ell n \triangleright P) \lor (\exists m < n. \ell m \triangleright \models^v_\text{le} P))
\]

By integrating the credit supply \( \ell n \), we avoid that Löb induction can be used to prove \( \models^v_\text{le} Q \) for any \( Q \). Each time we use the recursive case, we have to decrease the credit supply \( \ell n \). This can only be done through \textsc{SupplyDecr}, which requires us to give up (at least) one of our credits \( \ell n \), and can thus only be done finitely often. Hence the actual definition does not validate \( \triangleright P \vdash \models^v_\text{le} P \), but instead \( \triangleright P \vdash \ell 1 \vdash \models^v_\text{le} P \). The fact that the definition of \( \models^v_\text{le} P \) is meaningful is formalized by the following adequacy result:

**Lemma 5.1.** Let \( \phi \) be a first-order proposition (not involving any of the separation logic connectives or modalities). If \( \ell n \vdash \models^v_\text{le} \phi \) holds in Iris, then \( \phi \) holds at the meta level.

\(^9\)To formally define this recursive definition, we can use a least fixpoint, a greatest fixpoint, or Iris’s guarded recursion fixpoint—it does not matter. In Iris, every recursive definition where all recursive occurrences are guarded by a later has a unique solution [Jung et al. 2018b] (a consequence of Banach’s fixed-point theorem for metric spaces).

\(^{10}\)Readers familiar with Iris can think of the resources we use here as elements of the resource algebra \( \text{Auth}(\mathbb{N}, +) \). The credits \( \ell n \) are the fragments and the credit supply \( \ell n \) is the authoritative element.
To prove this lemma, we allocate a credit supply of $\ell \cdot n$ (together with $\ell n$), and then use an induction on $n$ to make sure the recursion terminates.

### 5.2 Hoare Triples

We turn to the model of Hoare triples and the proof that they guarantee partial correctness. Iris’s Hoare triples (with and without later credits) are rather complicated, so we focus on a simplified version, and then outline the extensions required for the full-blown version. In the simplified version, we focus on the integration of the later elimination updates (e.g., for \textsc{LEUpdExec}) and the fresh later credit generation (e.g., for \textsc{PureStep}), but omit concurrency, masks, and time receipts.

Similar to Iris we define Hoare Triples as $\{P\} e \{v, Q\} \triangleq \Box(P \rightarrow \text{wp } e \{v, Q\})$. Here, $\text{wp } e \{v, Q\}$ is the \textit{weakest precondition} that is required for $e$ to terminate safely in a value $v$ satisfying $Q$, or 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, so we can use them multiple times.)

In the definition of $\text{wp } e \{v, Q\}$, we deviate from Iris to integrate later credits. Under the simplifying assumptions, one may understand our definition as follows:

$$\text{wp } e \{v, Q\} \triangleq \left\{ \begin{array}{ll} \Rightarrow_{\text{le}} Q[e/\nu] & \text{if } e \in \text{Val} \\ \forall \sigma. \text{SI}(\sigma) \rightarrow \Rightarrow_{\text{le}} (\exists e', \sigma'. (e, \sigma) \rightarrow (e', \sigma')) * \\ & \rightarrow (\forall e', \sigma'. (e, \sigma) \rightarrow (e', \sigma') \rightarrow \\ & \ell 1 \rightarrow \Rightarrow_{\text{le}} \text{SI}(\sigma') * \text{wp } e' \{v, Q\} & \text{if } e \notin \text{Val} \end{array} \right. $$

This definition has two cases. In the first case, the value case, the weakest precondition is simply the postcondition $Q$ after an update. The second case, where $e$ is not a value, is more interesting. In that case, we assume an arbitrary heap $\sigma$ which satisfies the state interpretation $\text{SI}(\sigma)$. The state interpretation $\text{SI}(\sigma)$ ties the heap $\sigma$ in the weakest precondition to the assertions $\ell \rightarrow \nu$ in the program logic. After assuming $\text{SI}(\sigma)$, we get to execute an update and, subsequently, have to show two things. First, we have to show that $e$ can take a step in the current heap $\sigma$ (so it is not stuck). Second, we have to reason about all successor expressions $e'$ and successor heaps $\sigma'$. We have to show that, given one later credit $\ell 1$, we can execute an update, and subsequently we own the state interpretation for the new heap $\sigma'$ and the weakest precondition for the successor $e'$. (The later modality in the definition ensures that there is a unique solution to the equation.)

The integration of later credits reveals itself in two parts. First, we use our later elimination update $\Rightarrow_{\text{le}}$ instead of Iris’s original update modality $\Rightarrow$. Second, we add an additional assumption: in the expression case, we get to assume one later credit $\ell 1$ to prove the weakest precondition of the successor expression. This later credit is the one that is \textit{generated} when we take program steps (e.g., the one in \textsc{PureStep}).

At this point, we have seen almost all of the pieces to the later credits puzzle. The only thing that is missing is \textit{adequacy}, a proof that our modified Hoare triples actually imply partial correctness:

**Theorem 5.2.** Let $\phi$ be a first-order proposition. If $\vdash \{\text{True}\} e_1 \{v, \phi\}$ and $(e_1, \sigma_1) \rightarrow^n (e_n, \sigma_n)$, then either $(e_n, \sigma_n)$ can take another step, or $e_n \in \text{Val}$ and $\phi[e_n/\nu]$ holds at the meta level.

The proof of this theorem is similar to Iris’s adequacy proof [Jung et al. 2018b, Theorem 6]. The only difference is that we use Lemma 5.1 to allocate sufficiently many credits for the execution. We can use them to unfold the weakest precondition for $n$ steps and then show that, after $n$ steps, it implies the first-order proposition “$(e_n, \sigma_n)$ can take another step or $e_n \in \text{Val}$ and $\phi[e_n/\nu]$”.

---

\(^{11}\)Since the heap is orthogonal to later credits, we refer to [Jung et al. 2018b] for details on how $\ell \rightarrow \nu$ is tied to the heap.
Extensions. To explain the definition of Hoare triples, we have focused on a simplified setting omitting concurrency, masks, and time receipts. We now discuss what is required to remove these simplifications. For concurrency and masks, we can simply reuse everything that is already there in Iris. For the time receipts, we modify the time receipt extension of Jourdan [2021]. Jourdan makes the number of previous program steps $k$ available to the state interpretation $SI$, ties them to time receipts $\mathbf{X}$, and increases the number of laters that can be eliminated after every program step to $k$. Increasing the number of laters after every step is insufficient for our purposes (see §7), but we reuse the connection of time receipts $\mathbf{X}$ and the state interpretation $SI$ to control how many later credits are generated with each step: instead of one credit $\mathsf{£}1$, we assume $\mathsf{£}(k + 1)$ credits in the definition of the weakest precondition.

6 LIMITATIONS

Limited credit generation. One limitation that stands out is the limited generation of credits: We only generate a single credit per step and we had to add time receipts (in §2.3) to obtain the full prepaid invariant rule $\mathsf{InvPreOpen}$. Thus, one natural question to ask is: are these restrictions necessary, or would it be sound to generate more credits?

One answer to this questions is: we can generate significantly more credits by moving to a transfinitely step-indexed model. The historic root of this idea is the work of Svendsen et al. [2016]: they use a transfinite model to increase the number of “logical steps” (in our context later eliminations) that can be done with each physical step. We build on their idea and show that we can use a transfinite model to remove the credit generation restrictions. Specifically, we have defined later credits (as in §5.1) in Transfinite Iris [Spies et al. 2021]—a recently-developed, transfinitely step-indexed variant of Iris. We have then defined a modified version of Hoare triples (without time receipts) which admits one additional rule:

$$
\begin{align*}
\text{CreditsPost} \quad & \quad (P) \in \{v. Q\} \quad e \not\in \text{Val} \\
\quad & \quad (P) \in \{v. Q * \mathsf{£}n\}
\end{align*}
$$

With $\text{CreditsPost}$, we can allocate an arbitrary number of credits in the postcondition of an expression. Thus, $\text{CreditsPost}$ generalizes $\text{ReceiptCredits}$, but drops the need for time receipts. (In fact, we have even obtained $\text{CreditsPost}$ for a transfinite form of later credits $\mathsf{£}\alpha$.)

However, moving from Iris to Transfinite Iris does not come for free: Due to its transfinite model, Transfinite Iris does not support Iris’s commuting rules $\text{LaterExists}$ and $\text{LaterSep}$. These commuting rules are widely used in existing Iris proofs (e.g., proofs involving logical atomicity), and it is not clear yet whether all of those existing Iris proofs can be salvaged in Transfinite Iris. Unfortunately, it is inconsistent to have both the commuting rules and significantly increased later credit generation in the same logic. If we have the commuting rule $\text{LaterExists}$ and the rule $\text{CreditsPost}$, then proofs in our logic are no longer meaningful. For example, we can prove $\{\ell \leftrightarrow 0\} !\ell \{v. \text{False}\}$. We give a formal proof of this result in the supplementary material [Author(s) 2021]. The gist is the following: As Spies et al. [2021] have noted, in a logic with $\text{LaterExists}$ and $\text{Löb}$, the proposition $\exists n : N. \triangleright^n \text{False}$ is derivable. To prove $\{\ell \leftrightarrow 0\} !\ell \{v. \text{False}\}$, we can eliminate the existential quantifier in $\exists n : N. \triangleright^n \text{False}$ and obtain $\triangleright^n \text{False}$. Using $\text{CreditsPost}$, we can then generate $\mathsf{£}n$ credits and eliminate the $n$ laters guarding False, giving us False in the postcondition.

\[\text{For the exact details on the definition, we refer to the supplementary material [Author(s) 2021]. The gist is that Transfinite Iris has an additional property which Iris does not enjoy: the existential property, which allows us to hoist the witnesses of existential quantifiers out of the logic. Leveraging the existential property, we can existentially quantify over the number of credits that are generated per step, which allows us to derive the new rule.}\]
Opening prepaid invariants around updates. In §2.3, we have seen that later credits can be used to construct invariants which can be opened around Hoare triples without a guarding later. Naturally, this begs the question: what about opening prepaid invariants around updates (e.g., see InvOpenUpdate). Even with prepaid invariants, the guarding laters remain in the rule. This is not by accident: Krebbers et al. [2017a] have shown that an invariant opening rule without laters for updates is not sound. Since logically atomic updates are defined solely in terms of updates, we also do not inherit a later-free rule for LAInvOpen from prepaid invariants. Nevertheless, as we have seen in §4, later credits still bring significant benefits to logical atomicity proofs.

Backwards compaibility. The addition of later credits invalidates some rules about the interaction of so-called plain propositions and updates. These rules have been added as part of the development of runST [Timany et al. 2018], but are rarely used in other Iris proofs. Aside from the rules about plain propositions, the addition of later credits is backwards compatible. In particular, the mere presence of later credits does not force users to reason about them—ordinary proofs still proceed as before and, in advanced cases, users have later credits at their disposal. To demonstrate backwards compatibility, we have ported two significant Iris developments: ReLoC and logical atomicity (including a plethora of examples) to a version of Iris with later credits.

7 RELATED WORK

Steel. Steel [Swamy et al. 2020; Fromherz et al. 2021] is a shallow embedding of concurrent separation logic in F∗, which extends the programming language of F∗ with an effect for concurrency. 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 it does 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), which allows 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 emphasize that “contextual refinement proofs are beyond what is possible in Steel” [Fromherz et al. 2021, Page 27].

Multiple later eliminations per step. One effect of later credits is that we can eliminate multiple laters per step: we can save a credit of one step and then eliminate two laters after the next step. We are not the first to enable eliminating multiple laters per physical step. The idea originally goes back to Svendsen et al. [2016]. They define a transfinitely step-indexed logical relation which allows an arbitrary number of logical steps (their equivalent of later eliminations) per physical step. Whereas Svendsen et al. work directly in a transfinitely step-indexed model (i.e., not in a logic), Spies et al. [2021] have subsequently shown how to achieve a similar effect in a step-indexed separation logic built atop a transfinite model.

As we have discussed in §6, the reasoning afforded by transfinite models—generating an arbitrary number of credits per step—is complementary to later credits. Unfortunately, as pointed out by Spies et al. [2021], transfinite models also bring along complications such as the loss of the commuting rules LaterExists and LaterSep. Since these rules are widely used throughout many existing Iris developments, it is not yet clear how to satisfactorily mitigate the loss of these commuting rules. (By providing a fourth option for later elimination, as seen in the example of prepaid invariants, later credits do help mitigate the loss of the commuting rules, but they do not cover all their use cases.) To sidestep the commuting-rule complications, we have opted to primarily focus in this work on the use of later credits in a finitely step-indexed model—albeit then with some restrictions on credit generation as a consequence.
For finitely step-indexed models, it is well-known that an arbitrary fixed number of laters $k$ can be eliminated immediately after each step. Jourdan [2021] has shown how to generalize this idea to make the number of possible later eliminations dependent on the execution length (e.g., after 4 steps, 4 laters can be eliminated after the next step). As discussed in §5.1, he introduces a form of ghost state based on time receipts—$n$—to reflect how many program steps have passed so far.

In our work, we make use of a variation of this mechanism (as seen in §2.3 and §5.1), but our key idea of turning the right to eliminate laters into an ownable resource is novel, complementary, and (for our applications) crucial: increasing the number of later eliminations that can be performed right after a physical step (as his time receipts allow) does not suffice for implementing either prepaid invariants (where we need to eliminate laters before a step) or unsolicited helping (where there are no physical steps around).

REFERENCES


Anonymous Author(s). 2021. Later credits Coq development. Anonymous supplementary material submitted with the paper.


