Building Blocks for Step-Indexed Program Logics

Thomas Somers
Radboud University Nijmegen
The Netherlands
thomas.somers@ru.nl

Jonas Kastberg Hinrichsen
Aalborg University
Denmark
jkhi@cs.aau.dk

Lennard Gäher
MPI-SWS
Saarland Informatics Campus, Germany
gaeher@mpi-sws.org

Robbert Krebbers
Radboud University Nijmegen
The Netherlands
mail@robbertkrebbers.nl

Abstract

Step-indexing and the *later modality* $\triangleright P$ are widely used in program logics. A key challenge in proofs in step-indexed logics is turning $\triangleright P$ into P, coined the *later elimination problem.* Later elimination cannot be done unconditionally, and has traditionally been linked one-to-one to the physical steps the program performs in the operational semantics. This one-to-one correspondence proved limiting in practice, and various techniques (*flexible step-indexing* and *later credits*) have been proposed to relax this correspondence.

Unfortunately, there exist many variations of these techniques with different features and proof rules. Moreover, integrating these techniques into a program logic for a specific domain (*e.g.*, crash safety or trace refinement) requires non-trivial proof engineering of the metatheory. Our goal is to consolidate this situation. We introduce the *physical-step modality*—a modular building block that enables designers of program logics to obtain all existing features and rules with little proof engineering effort. We integrate our modality into various projects in the Iris ecosystem (Actris, RefinedRust, Perennial, Trillium), and show that it unlocks new proof rules that these projects previously did not support. All our results are mechanized in the Rocq prover.

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

Keywords: Step-Indexing, Later Modality, Iris, Rocq

ACM Reference Format:
Thomas Somers, Jonas Kastberg Hinrichsen, Lennard Gäher, and Robbert Krebbers. 2026. Building Blocks for Step-Indexed Program Logics. In *Proceedings of the 15th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '26), January 12–13, 2026, Rennes, France.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3779031.3779095

1 Introduction

Step-indexing [3] is a technique for stratifying recursive definitions without positivity requirement. It is widely used in program logics (most notably Iris [22, 23, 25–28] and VST [2, 6] to model among others first-class locks [21], impredicative invariants [45], higher-order ghost state [22], Rust-style borrows [23], and protocols based on session types [18]), logical relations models for languages with higher-order references and recursive types [1, 54], and type systems that guarantee productivity of corecursive functions [8, 36].

A common way of employing step-indexing is by internalizing it into a logic through the *later modality* $\triangleright$ [4, 36]. The intuitive idea is that propositions are modeled as infinite sequences $P = P_0, P_1, P_2, \ldots$ that are increasingly 'refined' (they are *downwards closed*). Depending on the use-case, each P_i is just a truth value, a set of heaps, or it has a more complicated structure. The later modality shifts the sequence by one and adds the trivial element to the front. For the case of truth values we have $\triangleright P = \text{True}, P_0, P_1, P_2, \ldots$

An important (and sound) property is *later introduction* (P entails $\triangleright P$), but its converse, *later elimination* ($\triangleright P$ entails P), is unsound as it would trivialize the modality. This means that if one has the assumption $\triangleright P$ (which happens commonly if one unfolds a recursive definition or a construction that uses the later internally), one needs to do non-trivial proof work to obtain P without the guarding later. Spies et al. [44] coined this the *later elimination problem.*

In much of the original work on step-indexing, the *logical steps* (elements P_i in the infinite sequence P) are connected one-to-one to the *physical steps* performed in the operational semantics of the program subject to verification. Hence, at each moment the program takes a physical step in the operational semantics, one is allowed to eliminate exactly one guarding later from each assumption $\triangleright P$ and obtain P. This one-to-one connection between logical steps and physical steps often proved too limiting in practice, *e.g.*, if multiple laters need to be eliminated due to the use of nested constructions that all use the later internally. Therefore two extensions to relax this connection have been proposed.

Flexible step-indexing [33] relaxes the restriction of removing exactly one later per physical step. At each physical step it allows one to eliminate a number of laters proportional to the number of physical steps performed so far. *Later credits* [44] allow one to eliminate laters at other moments than physical steps. They provide an ownable resource (in the separation-logic sense) $£\, n$ that gives the permission to eliminate n laters. During each physical step, one obtains one *later credit* $£\, 1$, which can be spent at any moment to eliminate a later (even when no physical step is in sight). Later credits can be combined with flexible step-indexing allowing one to obtain multiple later credits per physical step.

An orthogonal solution is *transfinite step-indexing* [46], part of Transfinite Iris [43], where propositions are modeled as sequences indexed by ordinals. Transfinite step-indexing has seen limited use in practice because it is incompatible with widely-used distributivity rules of the later modality. Indeed, the vast majority of current projects use step-indexing based on natural numbers, which is the focus of this paper. (See § 7 for a discussion on transfinite step-indexing.)

Problem and goal. Flexible step-indexing, later credits, and combinations thereof, are widely used in the Iris ecosystem. Yet, there is no canonical version—a number of variations have been developed, which all have different subsets of features and rules, and which are all modeled in different ways. This situation is unsatisfactory from a user's point of view. Different projects (which we discuss momentarily) made different choices, making it difficult to transfer ideas and libraries from one project to another.

The situation is also unsatisfactory for the designers of (domain-specific) program logics. Adding support for flexible step-indexing or later credits requires non-trivial proof engineering, touching intricate details about step-indexing—the kind of details that Iris aimed to hide in the very first place. Iris 3.0 [27] proposed a concise approach to define weakest preconditions/Hoare triples in which step-indexing is hidden. This approach proved effective in the design of domain-specific program logics for *e.g.,* continuations [51], effect handlers [9, 10, 55], crash safety [7], trace refinement [52] and non-interference [14, 17]. However, when integrating flexible step-indexing or later credits, details about step-indexing resurface. As a consequence many Iris projects do not support flexible step-indexing or later credits at all, or support different subsets of their rules.

The goal of this paper is to consolidate this situation by developing a modular building block for step-indexed program logics—called the **physical-step modality**—which has two key features. First, our modality ensures that designers of domain-specific program logics can focus on the features of their domain and get the features for step-indexing for free. Concretely, our modality can be used to give a concise definition of weakest preconditions. Details about step-indexing are encapsulated by the modality and hidden from the designer of the program logic. Proof engineering is eased as the weakest precondition rules (particularly those concerning

	⧗	⧖	£	$\mid\!\rightsquigarrow$	Investing	Custom WP
HeapLang	◐	●	●	○	○	○
RefinedRust [15]	●	◐	●	●	○	○
Perennial [7]	●	○	●	○	●	●
Trillium [52]	○	●	○	●	○	●
This paper	●	●	●	●	●	

Figure 1. Comparison of features in key projects (◐ ≈ 'partial support'; custom WP ≈ a domain-specific version of weakest preconditions is used instead of Iris's standard version).

step-indexing) and the *adequacy theorem* (which says that the program logic implies the desired program property, *e.g.,* safety) can be derived from generic rules of our modality. Second, our modality provides the union of all known rules for flexible step-indexing and later credits, making it easier to transfer ideas and libraries between different projects.

Supported features. Our physical-step modality provides the union of all features for flexible step-indexing and later credits that we know of in the Iris ecosystem, see Figure 1. To explain these features, we first explain some technical details. To support flexible step-indexing (with and without later credits), the program logic needs a mechanism to keep track of the number of physical steps the program has performed so far—which is a-priori a non-local property. Time receipts [35] (originally used for complexity analysis) provide a way to *locally* keep track of a lower bound.

There exist two versions of time receipts: *persistent* $⧖\, n$ and *additive* $⧗\, n$ receipts, satisfying $⧖\, n_1$ *and* $⧖\, n_2$ iff $⧖\, (n_1 \max n_2)$ and $⧗\, n_1$ *and* $⧗\, n_2$ iff $⧗\, (n_1 + n_2)$, respectively. Depending on the verification problem, one version might be a better fit than the other. As shown in Figure 1, most projects support one or the other, but not both. RefinedRust encodes persistent time receipts using additive ones, and Somers and Krebbers [42] do the converse in Actris [18, 20] (which is built on top of Iris's default language HeapLang). However, these encodings do not provide all known interaction rules (hence marked ◐). Mével et al. [35] (in the context of complexity analysis) proposed a rule to obtain a persistent 'snapshot' of an additive receipt, and Matsushita et al. [33] proposed a rule to 'add up' a persistent and additive receipt. At first sight, these rules appear incompatible, but we show that they can be obtained at the same time (§ 3.3).

Another difference between existing projects is the semantics of time receipts. Originally, they provide a lower bound on the number of physical steps that have been performed. The Perennial framework for crash safety [7] uses a different semantics to support *investing*, a feature allowing one to obtain exponentially more time receipts than steps performed. Perennial's metatheory thus employs an intricate way of counting the number of credits based on the physical steps.

Using our physical-step modality we are able to support all the aforementioned features in a combined setting (including the seemingly incompatible interaction rules), and encapsulate their details (such as the intricate counting argument of Perennial). We update the Rocq developments of all projects in Figure 1 to support the union of all features and simplify their metatheory.

Notably, the Trillium [52] framework for trace refinement only supported persistent time receipts. It was not immediately obvious whether Trillium was compatible with later credits at all. However, the consolidation of the step-indexing mechanisms provided by our physical-step modality allowed us to discover a way of integrating later credits into Trillium (§ 6.4). As an additional benefit, we give a generic definition of the *logical-step modality* ($|{\rightsquigarrow}\rangle$), of which specific versions have been developed in Aneris [16] (built on top of Trillium) and RefinedRust [15] (§ 6.6).

We believe our work opens the door to employ the discussed features in projects that previously have not used any of them, including but not limited to projects concerning effect handlers [9, 10, 55], continuations [51] and information-flow security [14, 17], as well as the VST-Iris project [31].

Contributions and outline. We start with a recap of step-indexed separation logic and the *later elimination problem* (§ 2). After that we turn to our contributions:

- We present a unified set of rules for persistent and additive time receipts, based on investing (§ 3).
- We present our key contribution—the **physical-step modality**. We show how it encapsulates step-indexing in the definition of weakest preconditions, its proof rules, and its adequacy theorem (§ 4).
- We present the model of the physical-step modality in terms of the Iris base logic (§ 5).
- We give an overview of the mechanization of our results in Rocq, how they generalize and ease the metatheory of the projects in Figure 1 (§ 6), and finally discuss limitations (§ 6.7).

We conclude with a discussion of related work (§ 7). All our results are mechanized in Rocq [41].

2 The Later Elimination Problem

We recap the *later elimination problem* through an overview of concurrent separation logic (§ 2.1), the traditional solutions to this problem (§ 2.2), and *later credits* (§ 2.3). We focus on Iris's default language HeapLang [23], but most Iris-based program logics for different domains have similar rules.

2.1 Concurrent Separation Logic

Separation logic propositions P assert ownership of resources; traditionally heaps of memory [38, 39]. The *points-to proposition* $\ell \mapsto v$ asserts exclusive ownership of a location ℓ with value v. Separation logic enables modular verification through the *separating conjunction* $P * Q$, which asserts that P and Q hold for *separate* resources. Hence $\ell_1 \mapsto v_1 * \ell_2 \mapsto v_2 \vdash \ell_1 \neq \ell_2$, and consequently $\ell \mapsto v_1 * \ell \mapsto v_2 \vdash \mathsf{False}$. The latter illustrates the concept of *exclusivity*: ownership of multiple instances of an exclusive resource entails a contradiction, which is often used to rule out impossible cases in proofs, as we will exemplify in § 2.3. Separation logic also features the *separating implication* $P \mathrel{-\!\!*} Q$ (a.k.a. "magic wand"), which interacts with the separating conjunction similarly to how regular implication interacts with regular conjunction, particularly, $P * (P \mathrel{-\!\!*} Q) \mathrel{-\!\!*} Q$.

As common in separation logic, we specify programs via the *weakest precondition* $\mathsf{wp}\ e\ \{\Phi\}$, which says that the program e cannot crash, and the postcondition $\Phi : \mathsf{Val} \to \mathsf{iProp}$ holds for any return value.[1] We write $\mathsf{wp}\ e\ \{w.\ P\}$ as sugar for $\mathsf{wp}\ e\ \{\lambda w.\ P\}$ and $\mathsf{wp}\ e\ \{P\}$ for $\mathsf{wp}\ e\ \{w.\ w = () * P\}$. Selected rules are shown in Figure 2. Rules in separation logic are entailments $P \vdash Q$, which express that the resources of P are included in Q. A separation logic proposition P is true, if it holds for all resources, *i.e.,* $\mathsf{True} \vdash P$. We use the inference rule notation $\frac{P}{Q}*$ to state that $P \mathrel{-\!\!*} Q$ is true, which is equivalent to $P \vdash Q$. We explicitly conjoin multiple premises using $*$ or $\wedge$, but if they are pure (*i.e.,* live in the Rocq meta logic) we use a space (in that case $*$ and $\wedge$ coincide).

The prototypical rules for the heap operations are WP-ALLOC, WP-LOAD, and WP-STORE. Allocation `ref` v yields exclusive ownership of a new memory cell $\ell \mapsto v$ (and has no precondition). Loading requires exclusive ownership of the memory location $\ell \mapsto v$, and returns the stored value v. Storing similarly requires exclusive ownership of the location, and yields ownership with the newly stored value $\ell \mapsto w$. To reason about concurrent programs, concurrent separation logic includes the parallel composition rule [5, 37]:

$$\begin{aligned}&\mathsf{wp}\ e_1\ \{\Phi_1\} * \mathsf{wp}\ e_2\ \{\Phi_2\} \mathrel{-\!\!*}\\ &\mathsf{wp}\ (e_1 \mid\mid e_2)\ \{(w_1, w_2).\ \Phi_1\ w_1 * \Phi_2\ w_2\}\end{aligned}$$

We can prove a weakest precondition for a parallel composition $(e_1 \mid\mid e_2)$ by verifying the threads e_1 and e_2 separately (as a consequence of the separating conjunction). The separating conjunction ensures that exclusive resources, such as points-to resources $\ell \mapsto v$, cannot directly be used in both threads. One way to share resources between threads is via a lock (mutex), whose proof rules are:

$$\begin{aligned}P &\mathrel{-\!\!*} \mathsf{wp}\ \mathit{new\_lock}\ ()\ \{lk.\ \mathbf{Lock}\ lk\ P\}\\ \mathbf{Lock}\ lk\ P &\mathrel{-\!\!*} \mathsf{wp}\ \mathit{acquire}\ lk\ \{P\}\\ \mathbf{Lock}\ lk\ P * P &\mathrel{-\!\!*} \mathsf{wp}\ \mathit{release}\ lk\ \{\mathsf{True}\}\end{aligned}$$

Here, we require P to be *exclusive*, *i.e.,* $P * P \mathrel{-\!\!*} \mathsf{False}$, to ensure a lock cannot be released twice.[2] The proposition $\mathbf{Lock}\ lk\ P$ captures that the value lk is a lock, guarding the

[1]Hoare triples are defined as $\{P\}\ e\ \{\Phi\} \triangleq \Box(P \mathrel{-\!\!*} \mathsf{wp}\ e\ \{\Phi\})$, where $\Box$ is Iris's persistence modality [23, §6].

[2]An alternative and slightly more complex approach to avoid double-release (possible in Iris) is the use of an exclusive `locked` token as in § 2.3.

Basic Separation Logic

WP-FRAME
$$\frac{P * \mathsf{wp}\ e\ \{\Phi\}}{\mathsf{wp}\ e\ \{w.\ P * \Phi\ w\}}*$$

WP-ALLOC
$$\mathsf{wp}\ \mathbf{ref}\ v\ \{\ell.\ \ell \mapsto v\}$$

WP-LOAD
$$\frac{\ell \mapsto v}{\mathsf{wp}\ !\,\ell\ \{w.\ w = v * \ell \mapsto v\}}*$$

WP-STORE
$$\frac{\ell \mapsto v}{\mathsf{wp}\ \ell := w\ \{\ell \mapsto w\}}*$$

WP-PURE-STEP
$$\frac{e_1 \xrightarrow{\text{pure}} e_2 \qquad \mathsf{wp}\ e_2\ \{\Phi\}}{\mathsf{wp}\ e_1\ \{\Phi\}}*$$

Invariants

INV-ALLOC
$$\frac{\triangleright P}{\Rrightarrow \boxed{P}}*$$

WP-UPD
$$\frac{(\Rrightarrow P) * (P \mathrel{-\!\!*} \mathsf{wp}\ e\ \{\Phi\})}{\mathsf{wp}\ e\ \{\Phi\}}*$$

INV-PERSISTENT
$$\mathsf{persistent}(\boxed{P})$$

INV-OPEN
$$\frac{\mathsf{atomic}(e) \qquad \boxed{P} * (\triangleright P \mathrel{-\!\!*} \mathsf{wp}\ e\ \{w.\ \triangleright P * \Phi\ w\})}{\mathsf{wp}\ e\ \{w.\ \Phi\ w\}}*$$

Laters and Later Credits

POINTSTO-TIMELESS
$$\mathsf{timeless}(\ell \mapsto v)$$

▷-TIMELESS
$$\frac{\mathsf{timeless}(P) \qquad \triangleright P}{\Rrightarrow P}*$$

▷-INTRO
$$\frac{P}{\triangleright P}*$$

WP-FRAME-▷
$$\frac{e \notin \mathsf{Val} \qquad \triangleright P * \mathsf{wp}\ e\ \{\Phi\}}{\mathsf{wp}\ e\ \{w.\ P * \Phi\ w\}}*$$

▷-EXISTS
$$\frac{\mathsf{inhabited}(A)}{\triangleright \exists x : A, \Phi\, x \mathrel{*\!\!*} \exists x : A, \triangleright \Phi\, x}*$$

▷-SEP
$$\triangleright(P * Q) \mathrel{*\!\!*} \triangleright P * \triangleright Q$$

LÖB-INDUCTION
$$(\triangleright P \rightarrow P) \vdash P$$

£-SPLIT
$$£\,(m+n) \mathrel{*\!\!*} £\,m * £\,n$$

£-USE
$$\frac{£\,1 * \triangleright P}{\Rrightarrow P}*$$

WP-PURE-STEP-£
$$\frac{e_1 \xrightarrow{\text{pure}} e_2 \qquad £\,1 \mathrel{-\!\!*} \mathsf{wp}\ e_2\ \{\Phi\}}{\mathsf{wp}\ e_1\ \{\Phi\}}*$$

Figure 2. Selected rules of Iris.

lock payload P. The **Lock** $lk\,P$ proposition is *persistent*, denoted persistent(**Lock** $lk\,P$), which means that we can freely duplicate it, *i.e.,* **Lock** $lk\,P \mathrel{-\!\!*}$ **Lock** $lk\,P\ *$ **Lock** $lk\,P$, and thus share it between threads. The specifications say that we can allocate a lock, by giving up the payload P, access the payload P by acquiring the lock, and release the lock by giving the payload P back. Let us show the lock in action:

$$\begin{array}{l}
\mathbf{let}\ \ell = \mathbf{ref}\ 0,\ lk = new\_lock\ ()\ \mathbf{in} \\
\left(\begin{array}{l} acquire\ lk; \\ \ell := !\,\ell + 1; \ell := !\,\ell + 1; \\ release\ lk \end{array} \middle\| \begin{array}{l} acquire\ lk; \\ \ell := !\,\ell + 1; \ell := !\,\ell + 1; \\ release\ lk \end{array}\right); \\
acquire\ lk;\ \ \mathbf{assert}\ (\mathtt{is\_even}\ !\,\ell);\ \ release\ lk
\end{array}$$

The crux of verifying that the **assert** cannot fail is to choose the right lock payload. We use $P \triangleq \exists x.\ \ell \mapsto x * is\_even\,x$. The payload holds for the initial state $\ell \mapsto 0$, it is preserved by the critical sections of the two threads, and it lets us verify the **assert**. As **Lock** $lk\,P$ is persistent, we can duplicate it and use it in the proof of both threads. An important observation is that inside the critical section we do not have to preserve the payload, but only need to restore it when releasing the lock, allowing us to increment the counter in multiple steps.

2.2 The Later Elimination Problem

We now explain the *later elimination problem* and three solutions to it, by verifying that a spin-lock implementation satisfies the aforementioned lock rules:

$$\begin{aligned}
new\_lock\ () &\triangleq \mathbf{ref\ false} \\
acquire\ lk &\triangleq \mathbf{if\ CAS}\ lk\ \mathbf{false\ true\ then}\ ()\ \mathbf{else}\ acquire\ lk \\
release\ lk &\triangleq lk := \mathbf{false}
\end{aligned}$$

The lock is implemented as a reference, initialized with **false**, signifying the unlocked state. Lock acquisition is a CAS-loop (compare-and-swap) that tests the state of the lock. If it is **false** (unlocked), the CAS sets it to **true** (locked), and *acquire* terminates; otherwise *acquire* loops. Lock release sets the lock to **false** (unlocking it). To prove the lock specification, we should define the lock predicate **Lock** $lk\,P$ to capture the invariant state of the lock:

$$\mathbf{Lock}\ lk\,P \triangleq \boxed{\exists b.\ lk \mapsto b * \mathbf{if}\ \neg b\ \mathbf{then}\ P}$$

A lock lk is a reference, storing a boolean value $lk \mapsto b$, s.t. when b is **false** (locked), the lock holds the payload P. Here, $\boxed{P}$ is Iris's *invariant assertion*, which expresses that P must hold invariably throughout the remainder of the program. Let us discuss the key rules of invariants.[3] We can freely turn any owned resource into an invariant (INV-ALLOC), allowing us to prove lock allocation. In INV-ALLOC Iris's *update modality* $\Rrightarrow P$ captures that we can obtain P through an update to ghost state. This modality can be eliminated when the goal is a weakest precondition (WP-UPD). Invariants are persistent (INV-PERSISTENT), which implies that **Lock** $lk\,P$ is persistent (and thus duplicable). We can momentarily access the invariant resources P during an atomic step (INV-OPEN). For example, during the CAS step, we can access the reference held by the invariant, and update it to a new value.

To avoid unsoundness related to the self-referential paradoxes [23, §3.4], INV-OPEN guards the invariant proposition with a *later* (modality) $\triangleright P$. To use the proposition after opening the invariant, we thus have to eliminate the later, begetting the *later elimination problem*. We focus on three solutions, and how they are used to prove the specification for lock acquisition: (1) distributivity rules, (2) timeless propositions, and (3) taking physical steps.

[3]For soundness, Iris invariants also involve *invariant masks* $\mathcal{E}$ and *namespaces* $\mathcal{N}$, which we omit for brevity's sake, since their use is standard.

In the proof of *acquire*, we need to obtain $lk \mapsto b$ (without later) to verify the CAS operation. To do so, we first open the invariant $\mathsf{Lock}\ lk\ P$ using INV-OPEN to obtain the assumption $\triangleright(\exists b.\ lk \mapsto b * \mathtt{if}\ \neg b\ \mathtt{then}\ P)$. We then use solution (1)—the distributivity rules $\triangleright$-EXISTS and $\triangleright$-SEP—to obtain $\triangleright\, lk \mapsto b$ and $\triangleright\, \mathtt{if}\ \neg b\ \mathtt{then}\ P$. To proceed, we use solution (2) of *timeless propositions*. A proposition is timeless if it does not depend on the step-index in the model [23, §5.7], or syntactically speaking, if it does not contain invariants, laters, or weakest preconditions. The key example of a timeless proposition is $\ell \mapsto v$ (POINTSTO-TIMELESS). Timeless propositions can have their later eliminated if the goal is a weakest precondition ($\triangleright$-TIMELESS, WP-UPD)—we hence obtain $lk \mapsto b$. We proceed by case analysis on b, and apply the rule for CAS (elided for brevity). In the success case (where $b = \mathtt{false}$, now set to $\mathtt{true}$) we get $\triangleright P$ from the invariant. Since P is not necessarily timeless we cannot use solution (2) to eliminate its later. We thus proceed with solution (3), *taking physical steps*, allowing us to eliminate at least one later at every physical step. This is captured by the WP-FRAME-$\triangleright$ rule, which we apply when resolving the if-statement.

To prove the failing case, we use LÖB-INDUCTION (applied at the beginning of the proof), which lets us assume that the *acquire* specification holds after at least one physical step (guarded by later). As before, we can eliminate the later from the induction hypothesis, using WP-FRAME-$\triangleright$. We elide the proof of *release* as it is similar to the proof of *acquire*.

2.3 Later Credits

We now demonstrate an additional, more recent, solution to the later elimination problem called *later credits* [44]. Later credits enable compositional reasoning about later elimination using an amortized approach that turns the right to eliminate n laters into an ownable separation logic resource $£\,n$. We can generate one later credit whenever a physical step (pure step, for brevity) is taken (WP-PURE-STEP-£), and use a later credit to eliminate one later (£-USE) at any point.

To understand the need for later credits, we consider a compositional approach to verifying the aforementioned (outer) lock specifications, on top of the following simpler (inner) lock specification without a payload P:

$$\begin{aligned}
&\mathrm{wp}\ \mathit{new\_lock}\ ()\ \{lk.\ \widehat{\mathsf{Lock}}\ lk\}\\
\widehat{\mathsf{Lock}}\ lk \mathrel{-\!\!*}\ &\mathrm{wp}\ \mathit{acquire}\ lk\ \{\mathsf{Locked}\ lk * £\,1\}\\
\widehat{\mathsf{Lock}}\ lk * \mathsf{Locked}\ lk \mathrel{-\!\!*}\ &\mathrm{wp}\ \mathit{release}\ lk\ \{\mathrm{True}\}\\
\mathrm{persistent}(\widehat{\mathsf{Lock}}\ lk) \quad &\mathrm{timeless}(\mathsf{Locked}\ lk)
\end{aligned}$$

Here, $\mathsf{Locked}\ lk$ expresses that we currently hold the lock, and prevents double-release. We discuss the color-coded later credit $£\,1$ momentarily. With these specifications we define the outer lock predicate with payload P as follows:

$$\mathsf{Lock}\ lk\ P \triangleq \widehat{\mathsf{Lock}}\ lk * \boxed{\mathsf{Locked}\ \ lk \vee P}$$

The proofs of the outer specifications of *new_lock* and *release* follow almost directly from the inner specifications. For *new_lock*, the inner specification yields a $\widehat{\mathsf{Lock}}$ predicate, which we use to allocate the invariant Lock of the outer lock. For *release*, we use the payload P (given in the precondition) to conclude that the outer invariant contains $\mathsf{Locked}\ lk$ (by exclusivity of P), swap the two, and use $\mathsf{Locked}\ lk$ to resolve the precondition of the inner release specification. For *acquire*, we verify the following outer specification:

$$\mathsf{Lock}\ lk\ P \mathrel{-\!\!*} \mathrm{wp}\ \mathit{acquire}\ lk\ \{\cancel{\triangleright}\ P\}$$

The crux is that proving this outer specification without the *undesired* color-coded later $\triangleright$ requires the color-coded later credit $£\,1$ in the inner specification. We need the later credit because we first have to apply the specification of *acquire*, yielding $\mathsf{Locked}\ lk$, and only then can we use INV-OPEN to swap with P by opening the outer invariant. However, we now obtain $\triangleright P$ *after* all physical steps have been taken, and thus cannot eliminate the later using the three methods from § 2.2. The problem is that the inner specification of *acquire* does not expose the fact that it takes at least one physical step. To solve this problem, we use later credits by extending the inner *acquire* specification with a later credit $£\,1$, which exactly captures that *acquire* takes at least one step. With the credit, along with the £-USE rule, we can eliminate the later, and prove the original outer specification.

3 Multiple Later Elimination

In § 2 we focused on eliminating a single later per physical step. There are cases where this is too restrictive, *e.g.*, when invariants and/or higher-order ghost state are nested. In some of these cases, later credits alone are sufficient, because they allow one to accumulate multiple credits from prior steps. However, there are cases where the number of laters to be eliminated grows proportionally to the number of physical steps taken, rendering later credits alone insufficient.

We explain the existing approaches to eliminating multiple laters per step, and how our approach is derived from them (§ 3.1). We then explain how two existing kinds of *time receipts* are used to keep track on the number of laters that can be eliminated (§ 3.2). We finally discuss how we support the union of all features from prior work (§ 3.3): generating multiple later credits, *investing* in Perennial [7], and the combination of both kinds of time receipts.

3.1 Our Approach to Flexible Step-Indexing

The traditional approach of allowing one later to be eliminated per physical step can be relaxed to multiple laters in different ways, as shown in Fig. 3.

A folklore extension is to eliminate a fixed number k rather than 1 later per step. This falls short for cases where the number of laters to be eliminated keeps growing, eventually surpassing k. Matsushita et al. [33] realized that the number of laters per step can depend on the total of number physical

Traditional step-indexing	$1 \longrightarrow 1 \longrightarrow 1 \longrightarrow \ldots$
Folklore extension	$k \longrightarrow k \longrightarrow k \longrightarrow \ldots$
Flexible step-indexing [33]	$1 \longrightarrow 2 \longrightarrow 3 \longrightarrow \ldots$
More flexible (current Iris)	$f\,1 \longrightarrow f\,2 \longrightarrow f\,3 \longrightarrow \ldots$
Our approach	$f\,1 \longrightarrow f^2\,1 \longrightarrow f^3\,1 \longrightarrow \ldots$

Figure 3. Number of laters that can be eliminated per physical step (figure adopted and extended from Spies et al. [44]).

steps taken, leading to *flexible step-indexing*. In the current version of Iris, this approach is generalized to a *generation function* f applied to the number of prior steps.

Similar to the current version of Iris, we parameterize our rules (Fig. 4) with a generation function f. However, the way we use the function f is different:[4] the number of laters eliminated in the next step is computed directly from the number of laters eliminated in the prior step by applying f. The use of an exponential function is inspired by Perennial [7] and allows us to support *investing*.

To prove the soundness theorem of our modality (Theorem 4.2) any generation function f may be chosen provided it satisfies two properties: (1) $f\ 0 = 0$, and (2) f is superadditive, *i.e.*, $\forall n, m.\, f\ (n + m) \geq f\ n + f\ m$. Our rules allow the generation function to be *over approximated*, as the rules for a smaller generation function may be derived from those of a larger one by weakening. As such, only property (1) is strictly necessary. Given any function f satisfying (1), a superadditive upper bound may be constructed from it.

The fact that the generation function f may be over approximated also enables modular proofs. In prior work, the choice of generation function was fixed by the program logic. As such, a user verifying a new module that required a larger generation function (for instance to use Perennial's investing technique) had to update the program logic and all other modules relying on the specific choice. We instead allow each module M to define its own generation function f_M, and define the top-level generation function f as the superadditive closure of the pointwise maximum. Our Rocq mechanization provides a method for constructing the top-level generation function in a convenient fashion (§ 6.1).

3.2 Both Kinds of Time Receipts Individually

A key challenge in flexible step-indexing is that the global step count is not directly accessible in the logic. Instead, separation logic resources called *time receipts* [35] can be used to track lower bounds on the number of steps. There are two variants: *persistent* time receipts (⧖n), which are duplicable, and *additive* time receipts (⧗n), which can be aggregated. Each form induces its own set of proof rules, with neither being strictly stronger than the other. Existing models of time receipts for later elimination are limited to either persistent or additive time receipts, but not both.

In Fig. 4 we present our unified set of rules that supports both kinds of receipts. Before exploring our rules for their interaction (§ 3.3), we focus on how both kinds of receipts are used individually. We explain this by verifying a queue inspired by message-passing channels in Actris [19, 42]. We remark that our queue is mainly used to build intuition and can be verified without laters or time receipts, but the original and more complicated version in Actris cannot (§ 6.5).

Queue example. In the previous section, we considered specifications for simple, non-recursive data structures. The same principles can be extended to recursive data structures, such as queues, for which a typical specification is:

$$\mathsf{wp}\ \mathit{new\_queue}()\ \{q.\, \mathbf{Q}\, q\, []\}$$

$$\mathbf{Q}\, q\, (v :: \vec{v}) \twoheadrightarrow \mathsf{wp}\ \mathit{dequeue}\ q\ \ \{w.\, w = v * \mathbf{Q}\, q\, \vec{v}\}$$

$$\mathbf{Q}\, q\, \vec{v} \twoheadrightarrow \mathsf{wp}\ \mathit{enqueue}\ q\ v\ \{\mathbf{Q}\, q\, (\vec{v} +\!\!+ [v])\}$$

$$\mathbf{Q}\, q_1\, \vec{v}_1 * \mathbf{Q}\, q_2\, \vec{v}_2 \twoheadrightarrow \mathsf{wp}\ \mathit{append}\ q_1\ q_2\ \{\mathbf{Q}\, q_1\, (\vec{v}_1 +\!\!+ \vec{v}_2)\}$$

The representation predicate $\mathbf{Q}\, q\, \vec{v}$ describes how the sequence of values $\vec{v}$ is represented in the heap. We focus on a linked-list implementation (Fig. 5). The queue itself is represented by a location q that holds pointers (h, t) to both the head and tail nodes, respectively. A constant-time enqueue operation is enabled by tracking the tail node. A possible representation predicate $\mathbf{Q}\, q\, \vec{v}$ is structured in two layers:

$$\mathbf{Q}\, q\, \vec{v} \triangleq \exists h, t.\, q \mapsto (h, t) * \mathbf{Q}_{\mathsf{rec}}\, h\, t\, \vec{v}$$

$$\begin{aligned} \mathbf{Q}_{\mathsf{rec}}\, h\, t\, \vec{v} \triangleq{} & (\vec{v} = [] * h = t * t \mapsto \mathsf{none}) \lor \\ & (\exists w, \vec{w}, h'. \\ & \quad \vec{v} = w :: \vec{w} * h \mapsto \mathsf{some}\ (w, h') * \triangleright \mathbf{Q}_{\mathsf{rec}}\, h'\, t\, \vec{w}) \end{aligned}$$

Its core is a recursive predicate $\mathbf{Q}_{\mathsf{rec}}\, h\, t\, \vec{v}$, describing the linked-list from head h to tail t containing the sequence $\vec{v}$. The outer predicate $\mathbf{Q}$ additionally asserts that the location q stores pointers to the current head and tail nodes. Defining the core $\mathbf{Q}_{\mathsf{rec}}\, h\, t\, \vec{v}$ requires recursion. For our simple queue we could have used structural recursion on the sequence $\vec{v}$. However, in more general settings such as Actris there might be no decreasing argument (§ 6.5). To mimic that use-case, we employ *guarded recursion*, where each recursive occurrence is guarded by a later modality, and apply Banach's fixpoint theorem [23, Thm 4] to prove that the fixpoint exists.

Using the representation predicate $\mathbf{Q}$, we can immediately verify the correctness of the *new_queue* and *dequeue* operations. The proof for *new_queue* follows by selecting the first branch of $\mathbf{Q}_{\mathsf{rec}}\, h\, t\, []$. For *dequeue*, the key step is to unfold $\mathbf{Q}_{\mathsf{rec}}\, h\, t\, (v :: \vec{v})$, which introduces a later guarding the recursive occurrence. This later is eliminated during the physical step that updates q to the new head pointer, allowing us to re-establish $\mathbf{Q}\, q\, \vec{v}$ in the postcondition.

Persistent time receipts. While the representation predicate $\mathbf{Q}$ suffices for verifying *new_queue* and *dequeue*, it is

[4]The actual formula to determine the laters per step from f is more complex than f^n, and is discussed in § 5.3.

⧖-ZERO
$$\Rrightarrow \text{⧖}0$$

⧖-PERSISTENT
$$\mathsf{persistent}(\text{⧖}n)$$

⧖-WEAKEN
$$n \geq m * \text{⧖}n \mathrel{-\!\!*} \text{⧖}m$$

⧗-ZERO
$$\Rrightarrow \text{⧗}0$$

⧗-COMBINE
$$\text{⧗}n * \text{⧗}m \mathrel{*\!\!-\!\!*} \text{⧗}(n+m)$$

⧗-SNAPSHOT
$$\text{⧗}n \mathrel{-\!\!*} \Rrightarrow \text{⧗}n * \text{⧖}n$$

⧗-⧖-INCR
$$\text{⧗}n * \text{⧖}m \mathrel{-\!\!*} \Rrightarrow \text{⧖}(n+m)$$

⧖-LATERN
$$\frac{e \notin \mathsf{Val} \qquad (\Rrightarrow \text{⧖}n) \wedge \big(((\Rrightarrow \triangleright \Rrightarrow)^{1+f\,n}\, P) * \mathsf{wp}\ e\ \{\Phi\}\big)}{\mathsf{wp}\ e\ \{w.\, P * \Phi\, v\}}*$$

⧖-INCR
$$\frac{e \notin \mathsf{Val} \qquad \text{⧖}n * \mathsf{wp}\ e\ \{\Phi\}}{\mathsf{wp}\ e\ \{w.\, \text{⧖}(f\,(n+1)) * \Phi\, w\}}*$$

⧗-USE
$$\frac{e \notin \mathsf{Val} \qquad \text{⧗}n * \mathsf{wp}\ e\ \{\Phi\}}{\mathsf{wp}\ e\ \{w.\, \text{⧗}(f\,n) * \pounds(f\,n) * \Phi\, w\}}*$$

⧗-PURE-STEP
$$\frac{e \xrightarrow{\text{pure}} e' \qquad (\Rrightarrow \text{⧖}n) \wedge \big(\text{⧗}(f\,(n+1)) * \pounds(f\,(n+1)) \mathrel{-\!\!*} \mathsf{wp}\ e'\ \{\Phi\}\big)}{\mathsf{wp}\ e\ \{\Phi\}}*$$

Figure 4. Our unified set of rules for time receipts.

$$\mathit{new\_queue}\ () \triangleq \mathtt{let}\ t = \mathtt{ref\,none}\ \mathtt{in}\ \mathtt{ref}(t,t)$$
$$\mathit{dequeue}\ q \triangleq \mathtt{let}\ (h,t) = {!\,q}\ \mathtt{in\,match}\ {!\,h}\ \mathtt{with\,none} \Rightarrow \mathtt{assert\,false} \mid \mathtt{some}\ (v,h') \Rightarrow q := (h',t);\ v\ \mathtt{end}$$
$$\mathit{enqueue}\ q\ v \triangleq \mathtt{let}\ (h,t) = {!\,q}\ \mathtt{in\,let}\ t' = \mathtt{ref\,none\,in}\ t := \mathtt{some}\ (v,t');\ q := (h,t')$$
$$\mathit{append}\ q_1\ q_2 \triangleq \mathtt{let}\ (h_1,t_1) = {!\,q_1}, (h_2,t_2) = {!\,q_2}\ \mathtt{in\,match}\ {!\,h_2}\ \mathtt{with\,none} \Rightarrow () \mid \mathtt{some}\ n \Rightarrow t_1 := \mathtt{some}\ n;\ q_1 := (h_1,t_2)\ \mathtt{end}$$

Figure 5. A linked-list implementation of a queue with constant-time enqueue and dequeue operations.

too weak for *enqueue*. The problem is that *enqueue* directly updates the tail node t, whose ownership is described by $\mathsf{Q_{rec}}\,h\,t\,\vec{v}$. Unfolding $\mathsf{Q_{rec}}\,h\,t\,\vec{v}$ introduces $|\vec{v}|$ laters guarding $t \mapsto \mathsf{none}$, but only a fixed number of physical steps are available before t is updated. Thus, eliminating one later and generating one later credit per step is insufficient.

To address this problem, we employ flexible step-indexing, and use *persistent time receipts* $\text{⧖}n$. This persistent time receipt is a permission asserting that later physical steps can eliminate at least $1 + f\ n$ laters. The proof rules for persistent time receipts are summarized in Figure 4. For now, we focus on the basic rules, omitting additive time receipts ⧗, later credits, and use the generation function $f\ n = n$.

The basic rules are as follows. First, the persistent time receipt 0 can always be allocated (⧖-ZERO). They can be duplicated (⧖-PERSISTENT), and weakened (⧖-WEAKEN). Persistent time receipts can be incremented after a step (⧖-INCR). Finally, the ⧖-LATERN rule is a generalization of WP-FRAME-▷ to eliminate multiple $1 + n$ laters around a step. Temporarily ignoring Iris's update modalities ($\Rrightarrow$), the rule states that given $\text{⧖}n$, we can eliminate $\triangleright^{1+n}$ from P by taking a step. The update modalities ($\Rrightarrow$) between the laters allow one to (among others) open nested invariants to establish P. We discuss the update modality around $\text{⧖}n$ and use of regular conjunction instead of separating conjunction in § 3.3.

Returning to the queue example, we verify *enqueue* by augmenting the representation predicate Q with time receipts as $\mathsf{Q}_{\text{⧖}}\ q\,\vec{v} \triangleq \mathsf{Q}\,q\,\vec{v} * \text{⧖}|\vec{v}|$. To prove *enqueue* we extract the tail pointer $\triangleright^{|\vec{v}|}\ t \mapsto \mathsf{none}$ from the queue predicate. During the $!\,q$ operation, we apply ⧖-LATERN to eliminate the laters from the necessary ownership assertion $t \mapsto \mathsf{none}$. After updating the tail, we reconstruct the queue representation predicate. We similarly increment the time receipt using ⧖-INCR to reflect the enqueued value. The proofs for *new_queue* and *dequeue* are extended in a similar way. We use ⧖-ZERO to establish the initial time receipt on allocation, and use ⧖-WEAKEN to decrease the time receipt when dequeuing.

A key benefit of persistent time receipts is that they can be layered on top of existing specifications. For instance, from WP-LOAD (Fig. 2), ⧖-LATERN and ⧖-INCR we can derive:

$$\text{⧖}n * (\triangleright^{n+1} P) * l \mapsto v \mathrel{-\!\!*} \mathsf{wp}\ !\,l\ \{w.\ w = v * \text{⧖}(n+1) * P\}$$

Similar specifications can be derived for other operations, including compound ones such as user-defined functions.

Additive time receipts. The representation predicate $\mathsf{Q}_{\text{⧖}}$ enables the verification of *enqueue*, but is insufficient for *append*. Specifically, the precondition contains time receipts $\text{⧖}\,|\vec{v}_1| * \text{⧖}\,|\vec{v}_2|$, whereas the postcondition requires $\text{⧖}\,(|\vec{v}_1| + |\vec{v}_2|)$. As *append* is constant-time, the ⧖-INCR rule is not sufficient to establish this postcondition.

Instead, we want to *independently* track time receipts for each queue, and then combine these bounds in the postcondition. This is achieved using *additive time receipts* $\text{⧗}n$ and augmenting the representation predicate to $\mathsf{Q}_{\text{⧗}}\ q\,\vec{v} \triangleq \mathsf{Q}\,q\,\vec{v} * \text{⧗}|\vec{v}|$, allowing us to verify all queue functions including *enqueue* and *append*. Additive time receipts can be combined using the ⧗-COMBINE rule, which is key to verifying *append*. They additionally satisfy a version of ⧖-LATERN with ⧖ replaced by ⧗, which can be derived from ⧖-LATERN (§ 3.3).

The caveat of additive time receipts is that they have no analog to ⧖-incr, as additive time receipts can only be generated once per physical step. Instead, incrementing an additive time receipt requires a separate rule for each operation, *e.g.*, ⧗-pure-step is specific for pure steps. As such, unlike persistent receipts, additive time receipts cannot be layered on top of existing specifications. For example, we cannot derive the following specification from wp-load:

$$l \mapsto v \twoheadrightarrow \text{wp } !\, l\ \{w.\ w = v * ⧗1\}$$

Instead, proving such specifications, both for primitive operations (*e.g.,* $!\,l$) and compound operations (*e.g.,* used-defined functions), requires proving a new WP rule in which the additive time receipt is explicitly included.

This section indicates that additive time receipts provide additional power over persistent ones, but are more difficult to use. Given this trade-off, we should provide users and designers of program logics with support for both kinds.

3.3 The Union of All Features

We now cover how our approach permits the union of all features from prior work (Figure 1).

Multiple later credits. So far we have focused on the interaction between time receipts and later elimination using the later modality. Time receipts can also be used to generate multiple later credits per step. The ⧗-pure-step rule extends wp-pure-step-£ to generate not 1 but multiple later credits per step, provided a suitable time receipt is supplied. The lower bound ⧖n in ⧗-pure-step determines the number of generated later credits, similar to the number of laters in ⧖-laterN. Moreover, additive time receipts can be given up for the duration of a step to generate their corresponding later credits (⧗-use). The ⧗-use rule can be layered on top of existing specifications, similar to the ⧖-laterN and ⧖-incr rules for persistent time receipts as described in § 3.2.

Investing. Additive time receipts in Perennial can be *invested*: giving up a single time receipt ⧗1 for a step (using ⧗-use) results in ⧗10 after the step (the number 10 is arbitrary, and could be any positive number). Investing is supported by picking any generation function f satisfying $f\ 1 > 1$. To obtain Perennial's rules we pick $f\ n = 10n$.

Perennial uses investing to circumvent the caveat of additive time receipts, namely to derive specifications that increase additive time receipts from specifications without. As a simple example, using the generation function $f\ n = 2n$, we can derive the following load specification from wp-load:

$$l \mapsto v * ⧗1 \twoheadrightarrow \text{wp } !\, l\ \{w.\ w = v * ⧗2\}$$

However, unlike ⧖-incr, this approach requires an initial time receipt ⧗1 (so the caveat is not circumvented entirely).

Both time receipts. Prior work on time receipts for step-indexing has modeled one type of time receipt in terms of the other using an invariant. For example, RefinedRust [15] models persistent using additive time receipts including the ⧗-⧖-incr rule (introduced by RustHornBelt [33]), but does not support the ⧖-incr rule. Conversely, Linking Actris [42] models additive time receipts using persistent time receipts, and does not support the ⧗-use rule.

Our theory supports both the ⧗-snapshot rule by Mével et al. [35] and the ⧗-⧖-incr rule. Individually, these rules allow turning a ⧗n into at most ⧖n. However, admitting both rules allows turning ⧗n into ⧖$2n$, seemingly making additive time receipts twice as large as persistent ones. Nevertheless, our model of time receipts (§ 5.1) permits both rules. Key to this interpretation is splitting the total time receipt count into two separate partitions, for additive and persistent time receipts respectively, and ensuring that the persistent partition remains the largest.

Our combination of persistent and additive time receipts makes it possible to derive some of the rules for ⧗ from those of ⧖, particularly, versions of ⧖-laterN and ⧗-pure-step with the premise ⧖n replaced by ⧗n. This is possible because there is an ordinary conjunction instead of a separating conjunction in the ⧖ version. The update before ⧖n allows one convert ⧗n into ⧖n using ⧗-snapshot, without losing ⧗n as an assumption in the remainder of the proof. More generally, our use of an ordinary conjunction allows one to use any Iris resource to establish the bound ⧖n without losing that resource in the remainder of the proof.

4 The Physical-Step Modality

Hoare triples are the heart of a program logic. In Iris they are defined in terms of the weakest preconditions (WP), which in turn, are defined using the Iris base logic [23, 27] rather than directly in Iris's step-indexed model. Using this indirection Iris aims to avoid explicit tracking of resources and step indices, thereby keeping the definition of WP and its metatheory modular and compact. However, we will show that Iris's current definition of WP with flexible step-indexing and later credits no longer adheres to this principle.

In this section we introduce our key contribution, the **physical-step modality**—a modular building block for defining WP and proving its adequacy theorem. We show that our modality restores a modular and compact definition of WP. We discuss the difference in structure of Iris's WP before and after using our modality (§ 4.1 and § 4.2). We then introduce the rules of our modality, and explain how they are used to derive the desired rules for WP (§ 4.3). Finally, we derive the adequacy theorem of WP from the soundness theorem of our modality (§ 4.4). In this section we focus on Iris's standard definition of WP, which we extend to several domain-specific program logics in § 6.

4.1 Current Weakest Precondition

Iris's weakest precondition is defined as a guarded recursive predicate in terms of the Iris base logic, which includes

The current version (later credits change in purple):

$$\mathsf{wp}\ e\ \{\Phi\} \triangleq \models\!\!\!\Rrightarrow \Phi\, e \qquad \text{if } e \in \mathsf{Val}$$

$$\mathsf{wp}\ e\ \{\Phi\} \triangleq \forall \sigma, n.\ S\,\sigma\, n \mathrel{-\!\!*} \models\!\!\!\Rrightarrow (\mathsf{red}(e,\sigma) \land (\forall e', \sigma'.\ ((e,\sigma) \to (e',\sigma')) \mathrel{-\!\!*} £(1 + f\ n) \mathrel{-\!\!*} (\models\!\!\!\Rrightarrow \triangleright \models\!\!\!\Rrightarrow)^{1+f\ n} \models\!\!\!\Rrightarrow (S\,\sigma'\ (n+1) * \mathsf{wp}\ e'\ \{\Phi\}))) \qquad \text{otherwise}$$

The version using our physical-step modality:

$$\mathsf{wp}\ e\ \{\Phi\} \triangleq \models\!\!\!\Rrightarrow \Phi\, e \qquad \text{if } e \in \mathsf{Val}$$

$$\mathsf{wp}\ e\ \{\Phi\} \triangleq \forall \sigma.\ S\,\sigma \mathrel{-\!\!*} (\models\!\!\!\Rrightarrow \mathsf{red}(e,\sigma)) \land (\forall e', \sigma'.\ ((e,\sigma) \to (e',\sigma')) \mathrel{-\!\!*} \models\!\!⧗\!\!\Rrightarrow (S\,\sigma' * \mathsf{wp}\ e'\ \{\Phi\})) \qquad \text{otherwise}$$

Figure 6. The definition of Iris's weakest precondition connective.

modalities such as the later $\triangleright P$ and update $\models\!\!\!\Rrightarrow P$. The current definition of weakest preconditions in Iris is depicted in Fig. 6, alongside a version using our physical-step modality. We omit details about masks and concurrency, as they are orthogonal to the physical-step modality.

First consider the original definition of weakest preconditions in Iris [23, 27] by ignoring details in purple regarding later credits and multiple later elimination. The proposition $\mathsf{wp}\ e\ \{\Phi\}$ asserts that if e is a value, then $\Phi\ e$ holds (after the opportunity to update the ghost state). If e is not a value, $\mathsf{wp}\ e\ \{\Phi\}$ consists of two parts: (1) progress $\mathsf{red}(e,\sigma)$, stating that e is not stuck, and (2) preservation, which asserts that the weakest precondition still holds after any step from e.

The weakest precondition also includes a *state interpretation* $S\,\sigma$, which relates the physical state σ (such as the heap) to resources in Iris (such as the assertion $\ell \mapsto v$). As Iris can be instantiated for different domains, the state, state interpretation and physical steps depend on the language. The original WP definition lines up with the motivation from Iris 3 [23, 27], being both modular and compact. Specifically, the single $\triangleright$ allows for stripping a later each step, the updates allow for updating ghost resources, and the state interpretation relates the physical state σ to resources in Iris.

Returning to the current model with later credits and multiple laters, the definition of weakest precondition is no longer compact. First, the single later modality is replaced by a complex iteration of laters and updates to support the elimination of multiple laters per step. Second, the state interpretation explicitly tracks the number of physical steps n, used to determine the number of laters and later credits. The definition is also no longer modular. It requires each language instantiation to define its own step counting mechanism in the state interpretation S, and prove the corresponding proof rules. These step counting mechanisms are typically not compatible with each other making it challenging to develop modular libraries that can be exchanged between logics for different domains and programming languages.

4.2 Weakest Precondition With Our Modality

The definition of WP using our physical-step modality (Fig. 6) is back to being modular and compact. The physical-step modality $\models\!\!⧗\!\!\Rrightarrow P$ asserts that the proposition P holds after the next *physical step*, similar to how $\triangleright P$ asserts that P holds after the next *logical step*. To prove P in $\models\!\!⧗\!\!\Rrightarrow P$, we can eliminate multiple laters, and use *all* the later credits and time receipts generated during the next physical step. Hence, exactly one such physical-step modality is included in WP for every physical step in the operational semantics.

Similar to the WP definition from Iris 3.0 (top of Fig. 6, without purple parts), there is no explicit tracking of step counts or resources. The presence of our physical-step modality alone is sufficient to support multiple later elimination using time receipts and later credits. This results in the same proof rules across language instances, and even different WP models that employ our physical-step modality.

Another change compared to the current WP definition (in blue) is to move the update into the conjunction. This change allows rules which use conjunction, such as ⧖-LATERN, to be derived from those of the physical-step modality. Because updates and conjunction do not commute, the new definition is weaker (*i.e.,* it is easier for a user to prove a WP), as it allows *different* updates to prove progress and preservation. Even so, as we will show in § 4.4, the new definition is strong enough to obtain the adequacy theorem. In § 6.7 we comment on the further consequences of making WP weaker.

4.3 Rules of Our Physical-Step Modality

The rules of the physical-step modality (Fig. 7) are similar to those of WP (Fig. 4), and are each annotated with the corresponding WP rule in parentheses. The remaining time receipt rules in Fig. 4 (⧖-ZERO–⧗-⧖-INCR) are readily provided by the theory of our physical-step modality. Similar to the rules of WP, the rules of our physical-step modality are parameterized by a generation function f (§ 3.1).

The rules of our physical-step modality correspond closely to the WP rules but with $\mathsf{wp}\ e\ \{\Phi\}$ replaced by $\models\!\!⧗\!\!\Rrightarrow P$. In addition to the time receipt rules, the physical-step modality can eliminate updates from assumptions (PSTEP-UPD). This rule is used to derive the corresponding WP rule (WP-UPD). Similarly, the PSTEP-LATERN rule is used to derive ⧖-LATERN, the PSTEP-⧖-INCR rule to derive ⧖-INCR, and the PSTEP-⧗-USE rule is used to derive ⧗-USE. The proofs of these derivations

pstep-upd (wp-upd)
$$\frac{(|\!\Rrightarrow P) \ast (P \mathrel{-\!\!\ast} |\!⧗\!\Rrightarrow Q)}{|\!⧗\!\Rrightarrow Q}\ast$$

pstep-laterN (⧖-laterN)
$$\frac{(|\!\Rrightarrow ⧖ n) \wedge \left(((|\!\Rrightarrow \triangleright |\!\Rrightarrow)^{1+f\,n}\ P) \ \ast |\!⧗\!\Rrightarrow Q\right)}{|\!⧗\!\Rrightarrow P \ast Q}\ast$$

pstep-⧖-incr (⧖-incr)
$$\frac{⧖ n \ast (|\!⧗\!\Rrightarrow P)}{|\!⧗\!\Rrightarrow ⧖(f(n+1)) \ast P}\ast$$

pstep-⧗-use (⧗-use)
$$\frac{⧗ n \ast (|\!⧗\!\Rrightarrow P)}{|\!⧗\!\Rrightarrow ⧗(f\,n) \ast £(f\,n) \ast P}\ast$$

pstep-intro (⧗-pure-step)
$$\frac{(|\!\Rrightarrow ⧖ n) \wedge \left(⧗(f(n+1)) \ast £(f(n+1)) \mathrel{-\!\!\ast} (|\!\Rrightarrow \triangleright |\!\Rrightarrow)^{1+f\,n}\ P\right)}{|\!⧗\!\Rrightarrow P}\ast$$

Figure 7. The rules of our physical-step modality.

are straightforward and follow directly from the rule of the modality and the model of WP in Fig. 6.

The remaining rule is pstep-intro, which states that when establishing P after the next physical step, we may first generate later credits and time receipts and eliminate laters. The number of later credits, time receipts and laters is determined by the time receipts available before the step. Since the physical-step modality corresponds to a step in the operational semantics in WP, the ⧗-pure-step rule is derived from this rule by introducing the physical-step modality in the model of WP corresponding to the pure step. Similarly, rules for introducing other primitive operations (such as load and store) that generate additive time receipts can be derived from pstep-intro.

4.4 Adequacy of Weakest Preconditions

A key theorem in Iris is the *adequacy* theorem, which states that proofs in Iris carry meaning in the meta-logic (*i.e.,* in Rocq). The adequacy theorem in Iris is stated as [23]:

Theorem 4.1 (Adequacy). *Let ϕ : Val $\rightarrow$ Prop be a pure (i.e., Rocq) predicate. If* wp e $\{\phi\}$ *and e executes in n steps to e', then either (1) e' can take another step, or (2) e' is a value v and $\phi\, v$ holds.*

The adequacy theorem turns proofs of weakest preconditions wp e $\{\phi\}$ into proofs of functional correctness of e, namely that e is safe (*i.e.,* does not get stuck), and if e terminates, then $\phi\, v$ holds for the resulting value v. The adequacy theorem consists of two parts, whose proofs are analogous. We focus on part (2). Using the physical-step modality, the proof of adequacy is relatively straightforward. First, we unfold the weakest precondition n times to end up in the value case. After simplifying, this results in:

$$S\,\sigma_0 \mathrel{-\!\!\ast} |\!⧗\!\Rrightarrow^n |\!\Rrightarrow \phi\, v$$

Here σ_0 is the initial state (*e.g.,* the empty heap), and $|\!⧗\!\Rrightarrow^n$ is the n-fold iteration of the physical-step modality. Iris allows us to obtain $|\!\Rrightarrow S\,\sigma_0$, and we are thus left with:

$$|\!⧗\!\Rrightarrow^n |\!\Rrightarrow \phi\, v$$

At this point, the soundness theorem of the physical-step modality (proven in § 5.3) can be applied to obtain $\phi\, v$:

Theorem 4.2 (Soundness of the physical-step modality). *If ψ : Prop is a pure proposition and $|\!⧗\!\Rrightarrow^n |\!\Rrightarrow \psi$, then ψ holds.*[5]

Compare the adequacy proof above to the current Iris WP model in Fig. 6. Unfolding WP gives:

$$S\,\sigma_0\ 0 \mathrel{-\!\!\ast} (|\!\Rrightarrow £x_i \mathrel{-\!\!\ast} (|\!\Rrightarrow \triangleright |\!\Rrightarrow)^{y_i})^n |\!\Rrightarrow \phi\, v$$

Here x_i and y_i are dependent on each step i. The proof then requires tedious bookkeeping of the x_i and y_i for each step to reshape the above into the form:

$$£x \mathrel{-\!\!\ast} (|\!\Rrightarrow \triangleright |\!\Rrightarrow)^{y}\ |\!\Rrightarrow \phi\, v$$

The result $\phi\, v$ can now be extracted using soundness of later credits [44]. As this bookkeeping needs to be redone for each custom WP, designers of a domain-specific logic may choose to opt out of later elimination features, for the sake of simplicity. Hence, by consolidating through our physical-step modality, we simplify the overhead associated with defining a custom WP where features for multiple later elimination are no longer opt-in—they are all readily available.

5 A Model of the Physical-Step Modality

In this section, we construct a model of our physical-step modality and prove its soundness theorem. The model construction builds on, and is inspired by that of later credits and the later elimination update [44] and consists of 3 parts. First, we define a model of time receipts that supports the time receipt rules in Fig. 4, including an additional *supply resource* which generates time receipts and bounds the existing time receipts (§ 5.1). Second, we define the physical-step modality modality using Iris's modalities, later credits, and our supply resource, and prove that it satisfies the rules in Fig. 7 (§ 5.2). Third, we prove soundness (Theorem 4.2). This proof depends on the bound provided by the supply resource, and the existing soundness theorem for later credits (§ 5.3).

5.1 The Time Receipt Supply

We introduce a *supply resource* `supply` $n_\bullet\ n_⧗$, a piece of Iris ghost state that governs the existing time receipts. Specifically, it states that any existing time receipt is at most $n_\bullet$,

[5]Including invariant masks, the last update is ${}^{\top}|\!\Rrightarrow^{\emptyset}$, which allows opening all invariants to establish ψ.

and that $n_{⧗}$ additive time receipts are currently *disabled*. Disabling corresponds to giving up additive time receipts for the duration of a physical step in the ⧗-USE rule.

The supply and time receipts themselves are modeled via Iris's mechanisms for ghost state. The construction is described below and is not technically complicated, but requires knowledge of Iris beyond the scope of this paper. This model ensures that time receipts satisfy the time receipt rules in Fig. 4, as well as the following rules for the supply:

SUPPLY-INIT
$$\mathrel{|\!\!\!\Rrightarrow} \exists \gamma.\ \mathbf{supply}_{\gamma}\ 0\ 0$$

SUPPLY-SNAPSHOT
$$\mathbf{supply}_{\gamma}\ n_{\bullet}\ n_{⧗} \mathrel{-\!\!*} ⧖_{\gamma}(n_{\bullet}/2)$$

SUPPLY-VALID
$$\mathbf{supply}_{\gamma}\ n_{\bullet}\ n_{⧗} * ⧖_{\gamma} n_{⧖} \mathrel{-\!\!*} n_{⧗} + n_{⧖} \le n_{\bullet}$$

SUPPLY-TR
$$\mathbf{supply}_{\gamma}\ n_{\bullet}\ n_{⧗} * ⧗_{\gamma} m \mathrel{*\!\!-\!\!*} \mathbf{supply}_{\gamma}\ n_{\bullet}\ (n_{⧗} + m)$$

SUPPLY-INCR
$$\mathbf{supply}_{\gamma}\ n_{\bullet}\ n_{⧗} \mathrel{-\!\!*} \mathrel{|\!\!\!\Rrightarrow} \mathbf{supply}_{\gamma}\ (n_{\bullet} + 2m)\ (n_{⧗} + m)$$

The SUPPLY-INIT rule creates a new empty supply with *ghost name* γ, which relates the supply and its corresponding time receipts. Since the physical-step modality uses a single supply—created in § 5.3 during the soundness proof—γ is generally left implicit (hidden using a type class in Rocq).

The model of $\mathbf{supply}\ n_{\bullet}\ n_{⧗}$ partitions $n_{\bullet}$ into the maximum persistent time receipt, and sum of additive time receipts. The SUPPLY-SNAPSHOT rule states that the persistent partition is at least half, so we can generate $n_{\bullet}/2$ persistent time receipts. Since additive and persistent time receipts are lower bounds of their partitions, SUPPLY-VALID states that no additive and persistent time receipt together exceed $n_{\bullet}$. The SUPPLY-TR rule disables and enables additive time receipts by moving them into and out of the supply. Finally, the SUPPLY-INCR rule creates new disabled additive time receipts by increasing both partitions by the same number.

Neither the SUPPLY-SNAPSHOT nor the SUPPLY-VALID rule provide a supply resource in the conclusion. For instance, instead of SUPPLY-SNAPSHOT, one might expect the following rule: $\mathbf{supply}_{\gamma}\ n_{\bullet}\ n_{⧗} \mathrel{-\!\!*} ⧖_{\gamma}(n_{\bullet}/2) * \mathbf{supply}_{\gamma}\ n_{\bullet}\ n_{⧗}$. When the conclusion of a rule in Iris is persistent, the assumptions are not consumed. Formally, given a rule $P \mathrel{-\!\!*} Q$ where $\mathrm{persistent}(Q)$, we have that $P \mathrel{-\!\!*} P * Q$. As such, the aforementioned rule is derivable from SUPPLY-SNAPSHOT.

Model of the supply. For Iris experts we summarize how **supply**, ⧖ and ⧗ are modeled using Iris's *view* camera [50]—a generalization of the *authoritative* camera [23, §6.3.3], inspired by the Views framework [11]. The authoritative and fragment elements of the view camera are related by a user-picked relation R (for the authoritative camera the relation is $R \triangleq \preccurlyeq$). A key property of the view camera over the authoritative camera is that fragments may be updated *without* owning the authoritative element, as long as the relation R remains valid. This key property of the view camera enables us to prove the ⧖-INCR and ⧗-SNAPSHOT rules without requiring ownership of the supply. For our supply and time receipts, the authoritative and fragment are $n_{\bullet}$ and the pair $(n_{⧗}, n_{⧖})$ respectively, using the relation:

$$R\ n_{\bullet}\ (n_{⧗}, n_{⧖}) \triangleq \exists p_{⧗}, p_{⧖}.\ n_{\bullet} = p_{⧗} + p_{⧖} \land p_{⧗} \le p_{⧖} \land \\ n_{⧗} \le p_{⧗} \land n_{⧖} \le p_{⧖}$$

The relation R partitions time receipts into an additive ($p_{⧗}$) and persistent ($p_{⧖}$) part, ensuring that the persistent part is largest, and that the fragment parts $n_{⧗}, n_{⧖}$ lie within the corresponding partitions. By the key property, the supply is not necessary to change the partitioning for the ⧖-INCR rule, as $n_{\bullet}$ remains the same. The time receipts $⧗n$ and $⧖n$ are modeled as owning a fragment of n of that time receipt, and 0 of the other. Finally, $\mathbf{supply}_{\gamma}\ n_{\bullet}\ n_{⧗}$ is modeled as owning the authoritative part and $n_{⧗}$ additive time receipts.

5.2 A Model of the Physical-Step Modality

We now define the physical-step modality using the supply, later credits, and existing Iris modalities as follows:

$$\mathrel{|\!⧗\!\!\Rrightarrow} P \triangleq \forall n_{\bullet}, n_{⧗}.\ \mathbf{supply}\ n_{\bullet}\ n_{⧗} * £\ (f\ (1 + n_{\bullet} - n_{⧗})) \mathrel{-\!\!*} \\ (\mathrel{|\!\!\!\Rrightarrow} \triangleright \mathrel{|\!\!\!\Rrightarrow})^{1 + f\ n_{\bullet}}\ \mathbf{supply}\ (n_{\bullet} + 2f(1 + n_{\bullet}))\ (f\ n_{⧗}) * P$$

The definition contains the supply $\mathbf{supply}\ n_{\bullet}\ n_{⧗}$ describing the maximum persistent time receipt $n_{\bullet}$ and number of disabled additive time receipts $n_{⧗}$. It provides the remaining later credits for the step (red) corresponding to the *enabled* time receipts, followed by eliminating one plus the maximum persistent time receipt (blue), allowing resource updates between each later. Finally, the updated supply and proposition P are established. The function f is the top-level generation function as described in § 3.1. Updating the supply creates $f(1 + n_{\bullet})$ new additive time receipts, increasing the supply by $2f(1 + n_{\bullet})$. This corresponds to the case in PSTEP-INTRO, where we use the maximum persistent time receipt $⧖n_{\bullet}$. Of these, a portion $f\ n_{⧗}$ corresponding to the previously disabled time receipts are kept disabled, whilst the rest are used to establish P.

This model and the earlier supply rules are sufficient to derive the rules in Fig. 7. Key to the derivation is that the supply bounds the size of *active* time receipts (SUPPLY-VALID), and in turn the number of laters and later credits.

The remaining problem is to distribute the new additive time receipts between PSTEP-INTRO and PSTEP-⧗-USE. This is done by *disabling* the time receipts used in PSTEP-⧗-USE, and enabling the corresponding time receipts after the step. In PSTEP-INTRO, the SUPPLY-VALID rule bounds n, and super-additivity of f (§ 3.1) is used to partition the later credits and additive time receipts between the active (n, from PSTEP-INTRO) and disabled ($n_{⧗}$) time receipts.

5.3 Soundness Theorem

Given the model of the physical-step modality, it remains to prove its soundness theorem (Theorem 4.2). This proof is done in two stages. First we remove the time receipt and physical-step modality abstractions and converting laters to later credits, and then we apply the existing soundness theorem for later credits [44, Lemma 5.3 & 5.5]:

Theorem 5.1 (Soundness of later credits)**.** *If ϕ is a pure (i.e., Rocq) proposition and $\pounds n \twoheadrightarrow \models\!\!\!\Rrightarrow \phi$ (in Iris), then ϕ holds.*

It remains to remove the time receipt and physical-step modality abstractions. First consider a single physical-step modality. In the model, the number of later credits and laters are determined by $\texttt{supply}\ n_\bullet\ 0$ (with all additive time receipts enabled). Converting laters to later credits using $\pounds$-USE, the number of later credits necessary for the step ($g_\pounds$), and maximum time receipt after the step ($g_\bullet$) are:

$$g_\pounds\ n_\bullet \triangleq (1 + f\ n_\bullet) + f\ (1 + n_\bullet) \qquad g_\bullet\ n_\bullet \triangleq n_\bullet + 2f\ (1 + n_\bullet)$$

Given initial supply $\texttt{supply}\ n_\bullet\ 0$, these functions determine the number of later credits necessary to unfold both single and multiple steps:

Lemma 5.2. *If $\texttt{supply}\ n_\bullet\ 0 * \models\!\!\blacktriangleright\!\!\!\Rrightarrow P$, then $\pounds(g_\pounds\ n_\bullet) \twoheadrightarrow \models\!\!\!\Rrightarrow \texttt{supply}\ (g_\bullet\ n_\bullet)\ 0 * P$.*

Proof. First unfold the definition of $\models\!\!\blacktriangleright\!\!\!\Rrightarrow$ and apply the initial supply. Next, use $\pounds$-USE to convert the laters into later credits (resulting in $g_\pounds\ n_\bullet$ in total) and $f\ 0 = 0$ (§ 3.1) to prove there are no disabled time receipts in the resulting supply. □

Theorem 5.3. *If $\texttt{supply}\ n_\bullet\ 0 * \models\!\!\blacktriangleright\!\!\!\Rrightarrow^k P$, then $\pounds\left(\sum_{i=0}^{k-1} g_\pounds\ (g_\bullet^i\ n_\bullet)\right) \twoheadrightarrow \models\!\!\!\Rrightarrow P$.*

Proof. Apply the prior lemma k times. □

A key part of this theorem is the formula $g_\pounds\ (g_\bullet^i\ n_\bullet)$, describing the number of later credits, hence laters, per step i for the initial supply $\texttt{supply}\ n_\bullet\ 0$. The final proof of soundness (Theorem 4.2) follows from creating the initial supply $\texttt{supply}\ 0\ 0$ using SUPPLY-INIT, then applying Theorem 5.3 and finally Theorem 5.1.

6 Mechanization and Applications

We mechanized the physical-step modality in Rocq using the Iris Proof Mode [26, 28], and migrated the projects in Figure 1. The mechanization is available on Zenodo [41]. We migrated the standard definition of WP in Iris and HeapLang, in the way described in this paper. For the other projects in Figure 1, the migration was similar. We discuss details related to the mechanization (§ 6.1), RefinedRust (§ 6.2), Perennial (§ 6.3), Trillium (§ 6.4), and Actris (§ 6.5). Finally, we give an overview of how we consolidated the *logical-step modality* used in RefinedRust and Trillium (§ 6.6), and conclude with limitations of our approach (§ 6.7).

6.1 Mechanization

A prerequisite for using our proof rules (Fig. 4) is selecting the right generation function f. As discussed in § 3.1, verifying different modules requires different generation functions, such as $f\ n = n$ for Actris (§ 6.5) or $f\ n = 10n$ to support Perennial's investing mechanism. Instead of hard-wiring this choice into the definition of WP, we let the user specify an appropriate function f_M for each module M, and obtain the top-level generation function f by taking the pointwise maximum of the functions f_M for all used modules.

Handling different generation functions for different modules resembles Iris's existing parameterization of *resource algebras* [49]. Each module individually defines the resource algebras it requires. When creating a closed proof in adequacy, the resource algebra is determined to be the product of resource algebras required by the used modules.

Our mechanization extends the existing Iris infrastructure such that each module specifies not only its required resource algebras, but optionally also its generation function. When constructing the product of resource algebras during adequacy, we also compute the top-level generation function. Consequently, different generation functions may be used as seamlessly as different resource algebras.

6.2 RefinedRust

RefinedRust [15] is an Iris-based verification tool for proving functional correctness of Rust programs using a semantic refinement type system. Like HeapLang, RefinedRust's program logic is based on the standard weakest precondition of Iris. RefinedRust modeled time receipts in a way that resulted in two WP rules for each primitive operation: one requiring time receipts, and one incompatible with them. The latter rule cannot be derived from the former. After migration, each primitive operation has a uniform proof rule using the physical-step modality, from which both prior variants can be derived. To more easily work with additive time receipts, RefinedRust provides a *logical step* modality ($\mid\!\!\leadsto P$), used throughout its semantic type system. We are able to recover this modality using our generalization presented in § 6.6.

6.3 Perennial

Perennial [7] is a program logic built on top of Iris for verifying safety in the context of crashes. A key ingredient is the extension of WP with a crash postcondition. In both the value and non-value case of the definition of this WP, there is a case split on whether the system has crashed. Both cases previously employed ad-hoc techniques for multiple later elimination. In our revised definition we use our physical-step modality only in the non-crash case. Instead of using an ad-hoc technique in the crash case, we noticed that this case does not need any multiple later elimination technique at all if we define their *later tokens* differently. With this change, we were able to recover all the original top-level rules of

Perennial. While we do not preserve the exact semantics of their original WP, the Perennial maintainers [40] confirmed that their original definition was not integral to the framework, and was only a means to make their top-level rules provable. The remainder of converting Perennial to use our physical-step modality is similar to HeapLang.

6.4 Trillium

Trillium is a program logic built on top of Iris for proving refinements between execution traces of a program and traces of a user-picked model [52]. Trillium's adequacy theorem says that one can obtain the refinement as a meta-level coinductive definition. This makes the adequacy proof atypical compared to other Iris-based program logics on three accounts. (1) The proof uses an intermediate definition, defined as a guarded fixpoint in Iris. The laters in this definition should align exactly with those in WP. (2) The proof uses a different soundness theorem for the update modality than Iris's default version, which permits iteratively eliminating updates from WP, necessary to establish a connection with the intermediate definition [53]. (3) The adequacy proof, and thus the whole of Trillium, is incompatible with later credits, as it uses conflicting lemmas about *plain* propositions [44].

In our conversion to the physical-step modality we had to consider all three points above. (1) We instrumented the intermediate definition with the functions $g_{\pounds}$ and $g_{\bullet}$ to ensure alignment of laters (§ 5.3). (2) We proved an alternative version of our soundness theorem (akin to [53]). (3) We observed that it is possible to make our version of Trillium compatible with later credits. We emphasize that the latter is not a consequence of using our physical-step modality, and could have also been achieved by changing the original definitions and proofs. Even so, we argue that the simplification of mechanisms provided by our physical-step modality is what allowed us to discover the possibility of re-enabling later credits in Trillium, emphasizing its benefits.

6.5 Actris

Actris [18–20, 42] is a framework for verification of message-passing programs based on session types built on top of Iris. A key ingredient is its channel ownership predicate $c \rightarrowtail p$, which is like our $\mathbf{Q}\, q\, \vec{v}$ predicate for queues (§ 3.2). Unlike the queue predicate, the argument p is not a first-order data type (such as a list of values), but a protocol. Protocols can be recursive and higher-order, necessitating the use of a guarded fixpoint (and laters) in the definition of protocols and the predicate $c \rightarrowtail p$. Consequently receiving on a channel (like *dequeue*) requires eliminating a number of laters n proportional to the channel's buffer size. Hinrichsen et al. [19] originally solved this problem by instrumenting the program with n times a **skip** instruction. Later, in their Rocq development, they removed the instrumentation by employing persistent time receipts. Somers and Krebbers [42] extended Actris with a *link* operation (like *append*) and employed additive time receipts, which they encode in terms of persistent time receipts. This encoding causes needless overhead (in particular, tracking an additional invariant mask in Iris), which our work removes. Gondelman et al. [16] developed a version of Actris for distributed networks on top of Aneris [29], and employ the logical-step modality discussed next.

6.6 The Logical-Step Modality

RefinedRust [15] and Aneris [16] independently developed an approach to handle multiple later elimination proof obligations, based on a logical-step modality $\mid\!\!\leadsto P$, capturing that resource P is available after a physical step has been taken. This intuition is made precise by the following rule:

$$\frac{e \notin \mathsf{Val} \qquad (\mid\!\!\leadsto P) * \mathsf{wp}\ e\ \{\Phi\}}{\mathsf{wp}\ e\ \{w.\, P * \Phi\, w\}}*$$

The logical-step modality is realized by internalizing the step counting, alongside ghost state enriched with time receipts. The versions of the modality in RefinedRust and Aneris are different. The former uses later credits and additive time receipts, while the latter uses laters and persistent time receipts. Our logical-step modality unifies both, and has a very simple definition in terms of our physical-step modality:

$$\mid\!\!\leadsto P \triangleq \forall R.\, (\models\!⧗\!\!\Rrightarrow R) -\!\!* \models\!⧗\!\!\Rrightarrow (R * P)$$

That is, P is frame-preserving (proven under any frame R), closed under the physical-step modality.

6.7 Limitations

In migrating the projects in Fig. 1 we determined the two limitations of using the physical-step modality. First, deriving time receipt rules with conjunction (*e.g.*, ⧖-laterN) requires weakening the definition of weakest precondition. Second, like the later modality, our modality is not a monad.

Weaker WP. A change to the WP definition (Fig. 6) is to move the update modality *into* the conjunction. This change results in a *weaker* definition, and consequently proofs of rules that have a WP as a premise need to be changed. For the WP rules this change was straightforward, as they involve a WP as a premise and the conclusion. The change to adequacy has been discussed in § 4.4 and § 6.4.

Our physical-step modality could be used without weakening WP, by replacing the conjunction in ⧖-laterN and ⧗-pure-step with a separating conjunction. This loses expressivity: one cannot use ⧗-⧖-incr or open invariants without closing them to establish the lower bound $⧖ n$ in ⧖-laterN, and ⧗-pure-step without affecting the other conjunct.

Not monadic. Many modalities in Iris, such as the update modality, are monadic (*i.e.*, transitive). This enables stripping the modality from an assumption, provided the same modality appear in the goal. For instance, to prove $(\models\!\!\Rrightarrow P) -\!\!* \models\!\!\Rrightarrow Q$, it suffices to prove $P -\!\!* \models\!\!\Rrightarrow Q$. Neither the later modality nor our physical-step modality have this property.

The original crash case of Perennial's WP used a modality similar to the physical-step modality that admits a rule of the form $⧗1 \ast (\models\!⧗\!\!\Rrightarrow \models\!⧗\!\!\Rrightarrow P) \mathrel{-\!\!\ast} \models\!⧗\!\!\Rrightarrow P$, where transitivity holds if a time receipt (later token in Perennial) is given up. Unlike their original modality, which *does not* generate time receipts, such a rule is unsound for our physical-step modality. Specifically, if $f\ 1 = 2$, such a rule and PSTEP-⧗-USE would result in $⧗2 \ast (\models\!⧗\!\!\Rrightarrow^{n} P) \mathrel{-\!\!\ast} \models\!⧗\!\!\Rrightarrow P$ for all n, and hence $⧗2 \ast \triangleright^{n} P \mathrel{-\!\!\ast} \models\!⧗\!\!\Rrightarrow P$, breaking soundness.

7 Related Work

Transfinite Iris. Transfinite step-indexing [46], which has been integrated into Transfinite Iris [43], allows the elimination of an unbounded number of laters at each physical step. It can be combined with later credits, allowing the generation of an unbounded number of later credits per step [44]. Unlike flexible step-indexing, this number does not depend on the number of physical steps performed so far, and as such, there is no need for time receipts to count steps. We thus believe that our modality is not applicable to transfinite step-indexing. However, we stress that transfinite step-indexing is not a silver bullet because it is incompatible with the distributivity rules $\triangleright$-SEP and $\triangleright$-EXISTS [43]. These rules are commonly used in Iris projects in practice, in particular, the Iris projects we discussed in § 6.

Steel and Pulse. Steel [13, 48] provides an embedding of concurrent separation logic in $F^{\star}$ [47], with features inspired by Iris. The way programs and proofs are written is different in Iris and Steel. In Iris, one states the program and then proves a weakest precondition (using tactics in Rocq), while in Steel proofs are written as annotations of the program. Consequently, in Steel, updates—such as opening and closing invariants or accessing ghost state—are written as program steps (ghost actions), which are erased during compilation. Steel supports dynamically-allocated invariants and first-order ghost state without step-indexing in the user-facing logic. Internally, this is achieved by treating ghost actions as steps, hence its model has a "later in disguise" [48, p18] at the point of ghost actions. Unlike Iris, Steel does not support higher-order ghost state and cannot be used beyond Hoare-style specifications, *e.g.,* contextual refinement [13, p27].

Pulse [12] is a more recent concurrent separation logic embedded in $F^{\star}$. Inspired by Iris, it adopts step-indexing and supports higher-order ghost state and impredicativity. Similar to Steel, Pulse represents updates as ghost functions, which are erased during compilation. However, unlike Steel and Iris, most program steps (and ghost functions) in Pulse do not allow one to eliminate a later. Thus, when opening an invariant, its content remains guarded by a later. Laters in Pulse are eliminated using later credits, which can be generated only by the *buy* ghost function that produces a single credit. As proofs and programs are intertwined in Pulse, the necessary number of credits, and hence buy ghost operations, can be determined. Consequently, there is currently no need for other operations to eliminate laters, as such *buy* operations can be inserted directly into the program.

Stratified propositions. A key reason for step-indexing in Iris is to support invariants and higher-order ghost state. Another approach, developed by Matsushita [32] and Lee et al. [30] is to employ *stratified propositions*, where invariants and higher-order ghost state are represented by a stratified datatype describing the syntax of the proposition, rather than the proposition itself, which would require impredicativity. This approach enables higher-order ghost state and dynamically-allocated invariants without step-indexing (and laters), which they employ to reason about liveness properties. Stratified propositions, however, come with a cost. They cannot support the impredicative features of Iris such as the encoding of logically atomic triples [24, 25]. Later work by Matsushita and Tsukada [34] extends stratified propositions to allow arbitrary separation logic propositions, by explicitly guarding them with a later. In that setting the multiple later elimination problem still applies for ghost state using impredicative features, so it would be interesting future work to see how our physical-step modality can be integrated.

Acknowledgments

We thank Amin Timany for discussions about Trillium, Tej Chajed, Upamanyu Sharma, and Joseph Tassarotti for discussions about Perennial, Ralf Jung for discussions about the later modality in Iris in general, and the anonymous reviewers for their insightful suggestions. The first and fourth authors are supported, in part, by ERC grant COCONUT (grant no. 101171349), funded by the European Union; and grant Cyclic Structures in Programs and Proofs (file number OCENW.XL.23.089), funded by the Dutch Research Council (NWO). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

References

[1] Amal Ahmed. 2004. *Semantics of Types for Mutable State.* Ph. D. Dissertation. Princeton University.

[2] Andrew W. Appel (Ed.). 2014. *Program Logics for Certified Compilers.* Cambridge University Press.

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

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

[5] Stephen Brookes. 2007. A semantics for concurrent separation logic. *TCS* 375, 1-3 (2007), 227–270. doi:10.1016/J.TCS.2006.12.034

[6] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. 2018. VST-Floyd: A Separation Logic Tool to Verify

Correctness of C Programs. *JAR* 61, 1-4 (2018), 367–422. doi:10.1007/S10817-018-9457-5

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

[8] Ranald Clouston, Ales Bizjak, Hans Bugge Grathwohl, and Lars Birkedal. 2015. Programming and Reasoning with Guarded Recursion for Coinductive Types. In *FoSSaCS (LNCS, Vol. 9034)*. 407–421. doi:10.1007/978-3-662-46678-0_26

[9] Paulo Emílio de Vilhena and François Pottier. 2021. A separation logic for effect handlers. *PACMPL* 5, POPL (2021), 1–28. doi:10.1145/3434314

[10] Paulo Emílio de Vilhena, Simcha van Collem, Ines Wright, and Robbert Krebbers. 2026. A Relational Separation Logic for Effect Handlers. *PACMPL* 10, POPL (2026). doi:10.1145/3776676

[11] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hongseok Yang. 2013. Views: compositional reasoning for concurrent programs. (2013), 287–300. doi:10.1145/2429069.2429104

[12] Gabriel Ebner, Guido Martínez, Aseem Rastogi, Thibault Dardinier, Megan Frisella, Tahina Ramananandro, and Nikhil Swamy. 2025. PulseCore: An Impredicative Concurrent Separation Logic for Dependently Typed Programs. *PACMPL* 9, PLDI (2025), 1516–1539. doi:10.1145/3729311

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

[14] Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2021. Compositional Non-Interference for Fine-Grained Concurrent Programs. In *S&P*. 1416–1433. doi:10.1109/SP40001.2021.00003

[15] Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. 2024. RefinedRust: A Type System for High-Assurance Verification of Rust Programs. *PACMPL* 8, PLDI (2024), 1115–1139. doi:10.1145/3656422

[16] Léon Gondelman, Jonas Kastberg Hinrichsen, Mário Pereira, Amin Timany, and Lars Birkedal. 2023. Verifying Reliable Network Components in a Distributed Separation Logic with Dependent Separation Protocols. *PACMPL* 7, ICFP (2023), 847–877. doi:10.1145/3607859

[17] Simon Oddershede Gregersen, Johan Bay, Amin Timany, and Lars Birkedal. 2021. Mechanized logical relations for termination-insensitive noninterference. *PACMPL* 5, POPL (2021), 1–29. doi:10.1145/3434291

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

[19] Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2022. Actris 2.0: Asynchronous Session-Type Based Reasoning in Separation Logic. *LMCS* 18, 2 (2022). doi:10.46298/LMCS-18(2:16)2022

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

[21] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. 2008. Oracle Semantics for Concurrent Separation Logic. In *ESOP (LNCS, Vol. 4960)*. 353–367. doi:10.1007/978-3-540-78739-6_27

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

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

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

[25] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*. 637–650. doi:10.1145/2676726.2676980

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

[27] Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The Essence of Higher-Order Concurrent Separation Logic. In *ESOP (LNCS, Vol. 10201)*. 696–723. doi:10.1007/978-3-662-54434-1_26

[28] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *POPL*. 205–217. doi:10.1145/3009837.3009855

[29] Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. 2020. Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems. In *ESOP (LNCS, Vol. 12075)*. 336–365. doi:10.1007/978-3-030-44914-8_13

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

[31] William Mansky and Ke Du. 2024. An Iris Instance for Verifying CompCert C Programs. *PACMPL* 8, POPL (2024), 148–174. doi:10.1145/3632848

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

[33] Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. 2022. RustHornBelt: A semantic foundation for functional verification of Rust programs with unsafe code. In *PLDI*. 841–856. doi:10.1145/3519939.3523704

[34] Yusuke Matsushita and Takeshi Tsukada. 2025. Nola: Later-Free Ghost State for Verifying Termination in Iris. *PACMPL* 9, PLDI (2025), 98–124. doi:10.1145/3729250

[35] Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2019. Time Credits and Time Receipts in Iris. In *ESOP (LNCS, Vol. 11423)*. 3–29. doi:10.1007/978-3-030-17184-1_1

[36] Hiroshi Nakano. 2000. A Modality for Recursion. In *LICS*. 255–266. doi:10.1109/LICS.2000.855774

[37] Peter W. O'Hearn. 2007. Resources, concurrency, and local reasoning. *TCS* 375, 1-3 (2007), 271–307. doi:10.1016/J.TCS.2006.12.035

[38] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *CSL (LNCS, Vol. 2142)*. 1–19. doi:10.1007/3-540-44802-0_1

[39] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*. 55–74. doi:10.1109/LICS.2002.1029817

[40] Upamanyu Sharma and Tej Chajed. 2025. Personal communication at 2025-08-25.

[41] Thomas Somers, Jonas Kastberg Hinrichsen, Lennard Gäher, and Robbert Krebbers. 2025. Rocq mechanization of "Building Blocks for Step-Indexed Program Logics". Zenodo. doi:10.5281/zenodo.17809073

[42] Thomas Somers and Robbert Krebbers. 2024. Verified Lock-Free Session Channels with Linking. *PACMPL* 8, OOPSLA2 (2024), 588–617. doi:10.1145/3689732

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

[44] Simon Spies, Lennard Gäher, Joseph Tassarotti, Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2022. Later credits: resourceful reasoning for the later modality. *PACMPL* 6, ICFP (2022),

283–311. doi:10.1145/3547631

[45] Kasper Svendsen and Lars Birkedal. 2014. Impredicative Concurrent Abstract Predicates. In *ESOP (LNCS, Vol. 8410)*. 149–168. doi:10.1007/978-3-642-54833-8_9

[46] Kasper Svendsen, Filip Sieczkowski, and Lars Birkedal. 2016. Transfinite Step-Indexing: Decoupling Concrete and Logical Steps. In *ESOP (LNCS, Vol. 9632)*. 727–751. doi:10.1007/978-3-662-49498-1_28

[47] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella-Béguelin. 2016. Dependent types and multi-monadic effects in F*. In *POPL*. 256–270. doi:10.1145/2837614.2837655

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

[49] The Iris Team. 2025. Global resource algebra management. https://gitlab.mpi-sws.org/iris/iris/-/blob/master/docs/resource_algebras.md

[50] The Iris Team. 2025. View Camera. https://gitlab.mpi-sws.org/iris/iris/-/blob/master/iris/algebra/view.v

[51] Amin Timany and Lars Birkedal. 2019. Mechanized relational verification of concurrent programs with continuations. *PACMPL* 3, ICFP (2019), 105:1–105:28. doi:10.1145/3341709

[52] Amin Timany, Simon Oddershede Gregersen, Léo Stefanesco, Jonas Kastberg Hinrichsen, Léon Gondelman, Abel Nieto, and Lars Birkedal. 2024. Trillium: Higher-Order Concurrent and Distributed Separation Logic for Intensional Refinement. *PACMPL* 8, POPL (2024), 241–272. doi:10.1145/3632851

[53] Amin Timany and Jonas Kastberg Hinrichsen. 2022. Iris Gitlab merge request: Strengthened soundness theorem. https://gitlab.mpi-sws.org/iris/iris/-/merge_requests/857

[54] Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2024. A Logical Approach to Type Soundness. *JACM* 71, 6 (2024), 40:1–40:75. doi:10.1145/3676954

[55] Orpheas van Rooij and Robbert Krebbers. 2025. Affect: An Affine Type and Effect System. *PACMPL* 9, POPL (2025), 126–154. doi:10.1145/3704841