Completeness of Iris-Based Program Logics

JOHANNES HOSTERT, ETH Zurich, Switzerland
ZICHEN ZHANG, New York University, USA
PUMING LIU, NYU Shanghai, China
SIMON ODDERSHEDE GREGERSEN, CISPA Helmholtz Center for Information Security, Germany
RALF JUNG, ETH Zurich, Switzerland
JOSEPH TASSAROTTI*, New York University, USA

Traditionally, proof systems such as program logics come with two core theorems: *soundness* and *completeness*. The role of soundness is obvious: we want to be sure that arguments carried out inside the logic actually lead to correct conclusions. Completeness complements that by ensuring that the logic does not limit expressivity: in principle, any correct result can be obtained within the confines of the logic. This result typically has to be stated *relative* to the completeness of the assertion logic that is used to reason about pre- and postconditions.

Over the past decade, the Iris framework has emerged as a widely used foundation for building separation logics. While Iris-based logics typically come with a soundness proof, none of them come with a proof of completeness. In fact, it is not entirely clear how to translate the typical recipe for relative completeness to Iris-based logics: these logics are higher-order, which means there is no separation of assertion logic and specification logic. Furthermore, the languages these logics reason about are also typically higher-order, enabling forms of recursion that go beyond a simple `while` loop (*e.g.*, Landin's Knot). In this paper, we present the first approach for establishing completeness of Iris-based program logics, and we show the generality of our methodology by applying it to a range of different logics described in prior work, including partial and total concurrent separation logics for a higher-order ML-like language, two quantitative logics (for bounding execution time and probabilistic errors), and a relational logic for proving refinement. All our results have been mechanized in the Rocq prover.

Additional Key Words and Phrases: Iris, Separation Logic, Completeness, Probability, Data Races

1 Introduction

Iris [33, 35, 38] is a higher-order separation logic framework that has been used for numerous program logics and verification projects. It provides reusable building blocks for constructing different variants of separation logic, with applications ranging from verification tools for C [47, 58], Rust [16, 48], OCaml [2, 24, 59], and Go [9], to reasoning about crash recovery [8], distributed execution [41, 60, 64], weak memory [13, 36, 50], resource consumption [53, 57], and probabilistic programs [1, 21, 22, 25, 26, 44], and semantic models of challenging type system features [15, 18–20, 23, 28, 32, 52, 65, 67]. All of these projects come with foundational soundness proofs, mechanized in the Rocq prover, that build on or adapt the core soundness proofs provided by the framework itself. These soundness results show that proofs of specifications about programs in the logic actually imply the intended properties about programs' executions.

Although the many applications of Iris-based logics provide some empirical evidence that these logics are highly expressive, there are no formal results that characterize the expressivity of these logics. In particular, it is unknown whether these logics are *complete*, that is, whether any program whose execution satisfies certain properties can be provably shown to do so by applying the proof

*Also affiliated with Amazon Web Services. This paper does not reflect the views of Amazon Web Services.

Authors' Contact Information: Johannes Hostert, johannes.hostert@inf.ethz.ch, ETH Zurich, Department of Computer Science, Zurich, Switzerland; Zichen Zhang, zichenzhang@nyu.edu, New York University, New York, New York, USA; Puming Liu, pl2559@nyu.edu, NYU Shanghai, Shanghai, China; Simon Oddershede Gregersen, gregersen@cispa.de, CISPA Helmholtz Center for Information Security, Saarbrücken, Germany; Ralf Jung, ralf.jung@inf.ethz.ch, ETH Zurich, Department of Computer Science, Zurich, Switzerland; Joseph Tassarotti, jt4767@nyu.edu, New York University, New York, New York, USA.

rules offered by these logics. Resolving this question is important because it tells us whether a logic is "missing" any rules, or whether the rules suffice to in principle verify any program.

For classical Hoare logic, completeness was long ago established by Cook [12]. This result is "relative" to the completeness of the assertion logic that is used for the rule of consequence. Subsequent works developed similar completeness results for logics for concurrency, such as the method of Owicki and Gries [56], shown complete by Owicki [55], and rely-guarantee reasoning [68]. Most recently, de Boer and Hiep [14] have shown how to adapt this style of approach to concurrent separation logic. However, these works consider languages and logics that are quite different from Iris. In prior work on completeness of concurrent program logics, the setting is a first order language, and the only source of looping behavior is a `while`-like loop construct. The logics are also first order with a strict stratification between assertion logic and specification logic. In contrast, Iris is often used to reason about higher-order languages that can encode Landin's knot [42], and the logic makes no distinction between assertions and specifications by supporting Hoare triples in pre- and postconditions, a crucial feature for reasoning about higher-order programs. As a result, existing results on relative completeness shed little light on the situation for logics like Iris.

This paper introduces a technique for proving completeness of Iris-based program logics. In doing so, we follow the usual approach of Iris: we provide reusable building blocks that can often be directly applied, and that can be further customized when needed. Concretely, we establish a core reusable lemma for obtaining completeness proofs for instances of Iris's language-agnostic default program logic. This nicely complements the existing language-agnostic soundness proof for said logic. We also demonstrate that the proof pattern behind this lemma generalizes to other Iris-based program logics that are not a direct instance of this framework.

Using our methodology, we prove, for the first time, that a number of Iris-based logics are complete (or can be made so by adding one or two missing proof rules). Our case studies include partial and total higher-order concurrent separation logics, two quantitative logics (for bounding execution time and probabilistic errors), and a relational logic for proving refinement. As expected, these proofs make use of the suite of reasoning principles that Iris provides (ghost state, invariants, and ownership). In particular, *Löb induction* [45], the fundamental primitive for recursive reasoning in step-indexed logics such as Iris, is powerful enough to let us show completeness even for languages with higher-order state. We demonstrate this with a series of case studies. All our results have been mechanized in the Rocq prover [63] using the Iris Proof Mode [37, 39].

We begin this paper in §2 by building an Iris-style program logic for a sequential language with higher-order state and proving it complete. In §3, we explain how Iris is used to reason about concurrent programs, and we extend the completeness proof to that setting. We continue with a series of completeness case studies: the logic for Iris' default language HeapLang (§4) and its total variant (§5); λ_{Rust}, which has a non-standard memory model to rule out data races (§6); a logic with time credits for reasoning about execution cost (§7); a probabilistic logic for reasoning about error bounds (§8); and finally a relational logic for proving refinement of concurrent higher-order programs (§9). We finish the technical contribution of the paper with a discussion of a general semantic condition that all complete Iris logics have to satisfy (§10). We then survey prior proofs of completeness for other program logics (§11) and conclude with a discussion about the significance of completeness (§12).

2 Warm-up: Completeness for a Sequential Language

In this section, we slowly build up to our main completeness result by first considering an Iris-style program logic for the simpler, sequential setting. Concretely, we define a language SeqLang, a call-by-value lambda calculus with integers and mutable higher-order state. The syntax and semantics can be found in Figure 1. We define a small-step operational semantics with the reduction relation

$$
\begin{aligned}
v, w \in \mathit{Val} &::= z \mid \ell \mid \lambda x.\, e \mid () && (z \in \mathbb{Z}, \ell \in \mathit{Loc}) \\
e \in \mathit{Expr} &::= v \mid x \mid \lambda x.\, e \mid e_1\, e_2 \mid e_1 \circledcirc_2 e_2 \mid \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \mid && (\circledcirc_2 \in \{+, -, =, \cdots\}) \\
&\quad\ \mathbf{ref}(e_1) \mid \mathbf{free}(e) \mid !\, e \mid e_1 \leftarrow e_2 \\
K \in \mathit{Ctx} &::= \bullet \mid e\, K \mid K\, v \mid e \circledcirc_2 K \mid K \circledcirc_2 v \mid \mathbf{if}\ K\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \mid \\
&\quad\ \mathbf{ref}(K) \mid \mathbf{free}(K) \mid !\, K \mid e \leftarrow K \mid K \leftarrow v \\
\sigma \in \mathit{State} &\triangleq \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Val}, \qquad \rho \in \mathit{Conf} \triangleq \mathit{Expr} \times \mathit{State}
\end{aligned}
$$

$$
\textsc{PureBeta}\ \ (\lambda x.\, e)\, v \rightarrow_{\mathsf{pure}} e[v/x]
\qquad
\textsc{PureOp}\ \ \frac{z_1 \circledcirc_2 z_2 = z_3}{z_1 \circledcirc_2 z_2 \rightarrow_{\mathsf{pure}} z_3}
\qquad \cdots \qquad
\textsc{BasePure}\ \ \frac{e_1 \rightarrow_{\mathsf{pure}} e_2}{(e_1, \sigma) \rightarrow_{\mathsf{base}} (e_2, \sigma)}
$$

$$
\textsc{BaseAlloc}\ \ \frac{\ell \notin \mathrm{dom}(\sigma)}{(\mathbf{ref}(v), \sigma) \rightarrow_{\mathsf{base}} (\ell, \sigma[\ell \leftarrow v])}
\qquad
\textsc{BaseFree}\ \ \frac{\ell \in \mathrm{dom}(\sigma)}{(\mathbf{free}(\ell), \sigma) \rightarrow_{\mathsf{base}} ((), \sigma \setminus \{\ell\})}
$$

$$
\textsc{BaseLoad}\ \ \frac{\sigma(\ell) = v}{(!\, \ell, \sigma) \rightarrow_{\mathsf{base}} (v, \sigma)}
\qquad
\textsc{BaseStore}\ \ \frac{\ell \in \mathrm{dom}(\sigma)}{(\ell \leftarrow v, \sigma) \rightarrow_{\mathsf{base}} ((), \sigma[\ell \leftarrow v])}
\qquad
\textsc{StepCtx}\ \ \frac{(e_1, \sigma) \rightarrow_{\mathsf{base}} (e_2, \sigma')}{(K[e_1], \sigma) \rightarrow (K[e_2], \sigma')}
$$

Fig. 1. Syntax and Semantics of SeqLang.

$\rightarrow$ using evaluation contexts K and a *base* reduction relation $\rightarrow_{\mathsf{base}}$. The definition of evaluation contexts induces a right-to-left evaluation order.

The step rules operate on configurations ρ, which are pairs consisting of an expression e and a heap σ. The heap is represented as a finite partial map from locations to values. The reduction of the pure, deterministic fragment of the language is described by the *pure* reduction relation $\rightarrow_{\mathsf{pure}}$, which can (using BasePure) be turned into a base reduction step that leaves the heap untouched. Note that Figure 1 omits some of the pure reductions for brevity. The rules for the load and store operations are standard. Deallocation removes the heap cell (which can then be re-used by a future allocation). For all of these operations, the reduction relation *gets stuck* if the location is not allocated. Allocation is the only non-deterministic operation in our language since it picks an arbitrary fresh location $\ell \notin \mathrm{dom}(\sigma)$.

2.1 A Sequential Separation Logic

Figure 2 defines a separation logic for SeqLang. This separation logic can be used to derive judgments of the shape $P \vdash Q$, where P and Q are separation logic assertions. Separation logic enriches propositional logic with the separating conjunction $*$ and separating implication $-\!\!*$, sometimes called the "magic wand". The separating conjunction $P * Q$ expresses ownership of P and Q *separately, i.e.*, P and Q hold for disjoint parts of the heap. Thus, in general, $P \nvdash P * P$. Most often, Iris-based separation logics are *affine* logics, meaning they allow the weakening rule $P * Q \vdash P$. The separating implication $-\!\!*$ is a version of implication that interacts with separating conjunction the same way that regular implication interacts with conjunction: $P \vdash Q -\!\!* R$ iff $P * Q \vdash R$. We do not give a full introduction to separation logic here but refer to Chapter 3 of Birkedal and Bizjak [5] for an expository introduction.

The assertion $\ulcorner\varphi\urcorner$ *embeds* an arbitrary meta-level proposition φ (*i.e.*, a Rocq `Prop`) into the syntax of the logic. In particular, if $\varphi \Rightarrow \varphi'$ holds at the meta-level then $\ulcorner\varphi\urcorner \vdash \ulcorner\varphi'\urcorner$. This means that the proof rules of the logic are not purely syntactic.

To turn our logic into a program logic, we first introduce the points-to connective $\ell \mapsto v$. This assertion expresses that the current heap stores the value v at location ℓ. Furthermore, this fact is *exclusive*, meaning it is not possible to own two points-to connectives for the same piece of state. This is also witnessed by the rule PointsToNe.

We reason about program executions using $\mathsf{wp}\ e\ \{\Phi\}$. Like all separation logic assertions, this is an assertion about the current program state: wp holds in some program state if executing e from that state executes safely, and if furthermore whenever e terminates with a value v, then $\Phi(v)$ holds for the final state. This property expresses a form of partial correctness as it does not guarantee termination. We discuss the definition of wp in §10, but for now we treat it as an abstract predicate that satisfies the rules in Figure 2. The later modality $\triangleright$ can be ignored for now.

The structural rule WpWand can be used to prove both monotonicity of the postcondition as well as the *frame rule*, the hallmark of separation logic:[1]

$$\mathsf{wp}\ e\ \{\Phi\} * P \vdash \mathsf{wp}\ e\ \{v.\ \Phi(v) * P\}$$

This rule is the key ingredient to *modular* reasoning about programs: If we have proven a specification $\mathsf{wp}\ e\ \{\Phi\}$ about the program e, then any assertion P on some unrelated state (that is disjoint from the state used in the verification) will not affect the execution. As such, P can be passed along to the postcondition to be used unchanged once e has finished executing.

The rule WpValue is the "base case" of the wp, since we just plug the value into the postcondition. The rule WpBind can be used to focus on a sub-expression in an evaluation context. All the remaining reasoning rules correspond to base reduction steps, with the rule WpPure re-using the pure reduction relation from Figure 1. The rules for the heap-manipulating steps are written in a "predicate transformer style" with Φ serving as a form of continuation. For example, the rule WpStore says that to reason about a store of w to location ℓ, we must own $\ell \mapsto v$ for some value v. In addition, we must show that the post-condition Φ would follow from having $\ell \mapsto w$, the updated points-to that reflects the result of the store. From this rule, it is straightforward to derive an alternate "direct style" version that looks like a typical Hoare triple: $\ell \mapsto v \vdash \mathsf{wp}\ \ell \leftarrow w\ \left\{v.\ \ulcorner v = ()\urcorner * \ell \mapsto w\right\}$.

There is no rule in Figure 2 for reasoning about loops as SeqLang does not have a primitive loop or recursion mechanism. However, since it is a higher-order language and supports higher-order state, recursion can be encoded in multiple ways. We postpone the discussion of how to reason about recursion until §2.3.

Before we consider the question of completeness, we verify two examples to demonstrate how the rules are to be used.

Example 1. We prove that the program $40 + (1 + 1)$ evaluates to the value 42. We start with the goal at the top and apply the rules one-by-one, noting what is left to show. This is how the rules are intended to be used, and the resulting order of proof steps follows the program's execution order.

[1]The notation $\mathsf{wp}\ e\ \{\Phi\}$ is sugar for $\mathsf{wp}\ e\ \{v.\ \Phi(v)\}$.

$$
\begin{array}{llr}
P, Q \in \mathit{iProp} ::= \ulcorner \varphi \urcorner \mid P \wedge Q \mid P \vee Q \mid \forall x.\, P \mid \exists x.\, P \mid \cdots & & (\varphi \text{ a meta-level proposition}) \\
\qquad \mid P * Q \mid P \mathrel{-\!\!*} Q \mid \ell \mapsto v \mid \mathsf{wp}\ e\ \{v.\, P\} \mid \triangleright P & & (\Phi, \Psi \text{ predicates } \mathit{Val} \to \mathit{iProp})
\end{array}
$$

PointsToNe
$$\ell_1 \mapsto v_1 * \ell_2 \mapsto v_2 \vdash \ulcorner \ell_1 \neq \ell_2 \urcorner$$

WpWand
$$\mathsf{wp}\ e\ \{\Phi\} * (\forall x.\, \Phi(x) \mathrel{-\!\!*} \Psi(x)) \vdash \mathsf{wp}\ e\ \{\Psi\}$$

WpValue
$$\Phi(v) \dashv\vdash \mathsf{wp}\ v\ \{\Phi\}$$

WpBind
$$\mathsf{wp}\ e\ \{v.\, \mathsf{wp}\ K[v]\ \{\Phi\}\} \dashv\vdash \mathsf{wp}\ K[e]\ \{\Phi\}$$

WpPure
$$\frac{e_1 \to_{\mathsf{pure}} e_2}{\triangleright\, \mathsf{wp}\ e_2\ \{\Phi\} \vdash \mathsf{wp}\ e_1\ \{\Phi\}}$$

WpFree
$$\ell \mapsto v * \triangleright \Phi(()) \vdash \mathsf{wp}\ \mathbf{free}(\ell)\ \{\Phi\}$$

WpAlloc
$$\triangleright \forall \ell.\, \ell \mapsto v \mathrel{-\!\!*} \Phi(\ell) \vdash \mathsf{wp}\ \mathbf{ref}(v)\ \{\Phi\}$$

WpLoad
$$\ell \mapsto v * \triangleright(\ell \mapsto v \mathrel{-\!\!*} \Phi(v)) \vdash \mathsf{wp}\ !\,\ell\ \{\Phi\}$$

WpStore
$$\ell \mapsto v * \triangleright(\ell \mapsto w \mathrel{-\!\!*} \Phi(())) \vdash \mathsf{wp}\ \ell \leftarrow w\ \{\Phi\}$$

Fig. 2. Separation Logic for SeqLang.

$$
\begin{array}{lll}
 & \vdash \mathsf{wp}\ 40 + (1+1)\ \{v.\, \ulcorner v = 42 \urcorner\} & \\
\Leftarrow & \vdash \mathsf{wp}\ 1 + 1\ \{w.\, \mathsf{wp}\ 40 + w\ \{v.\, \ulcorner v = 42 \urcorner\}\} & \text{by WpBind with } K := 40 + \bullet \\
\Leftarrow & \vdash \mathsf{wp}\ 2\ \{w.\, \mathsf{wp}\ 40 + w\ \{v.\, \ulcorner v = 42 \urcorner\}\} & \text{by WpPure} \\
\Leftarrow & \vdash \mathsf{wp}\ 40 + 2\ \{v.\, \ulcorner v = 42 \urcorner\} & \text{by WpValue} \\
\Leftarrow & \vdash \mathsf{wp}\ 42\ \{v.\, \ulcorner v = 42 \urcorner\} & \text{by WpPure} \\
\Leftarrow & \vdash \ulcorner 42 = 42 \urcorner & \text{by WpValue}
\end{array}
$$

Example 2. We prove that the program $(\lambda x.\ !\,x)\ (\mathbf{ref}(42))$ evaluates to the value 42 as well. This program involves mutable state, so a formal proof involves manipulating separation logic assertions; we omit those details in the proof outline shown below.

$$
\begin{array}{lll}
 & \vdash \mathsf{wp}\ (\lambda x.\ !\,x)\ (\mathbf{ref}(42))\ \{v.\, \ulcorner v = 42 \urcorner\} & \\
\Leftarrow & \vdash \mathsf{wp}\ \mathbf{ref}(42)\ \{w.\, \mathsf{wp}\ (\lambda x.\ !\,x)\ w\ \{v.\, \ulcorner v = 42 \urcorner\}\} & \text{by WpBind} \\
\Leftarrow & \ell \mapsto 42 \vdash \mathsf{wp}\ \ell\ \{w.\, \mathsf{wp}\ (\lambda x.\ !\,x)\ w\ \{v.\, \ulcorner v = 42 \urcorner\}\} & \text{by WpAlloc} \\
\Leftarrow & \ell \mapsto 42 \vdash \mathsf{wp}\ (\lambda x.\ !\,x)\ \ell\ \{v.\, \ulcorner v = 42 \urcorner\} & \text{by WpValue} \\
\Leftarrow & \ell \mapsto 42 \vdash \mathsf{wp}\ !\,\ell\ \{v.\, \ulcorner v = 42 \urcorner\} & \text{by WpPure} \\
\Leftarrow & \ell \mapsto 42 \vdash \ulcorner 42 = 42 \urcorner & \text{by WpLoad}
\end{array}
$$

Soundness. We have seen how the logic can be used to verify simple programs, and we have discussed that the wp encodes a partial correctness criterion. However, we still need to show that the logic is sound: proving a wp about a program should establish partial correctness of the program at the meta-level. To state this formally, we first define a meta-level predicate $\mathit{safe}_\varphi(e, \sigma)$:

$$
\begin{aligned}
\mathsf{red}(e, \sigma) &\triangleq \exists e', \sigma'.\, (e, \sigma) \to (e', \sigma') \\
\mathit{safe}_\varphi(e, \sigma) &\triangleq \forall e', \sigma'.\, (e, \sigma) \to^* (e', \sigma') \implies (\exists v.\, e' = v \wedge \varphi(v)) \vee \mathsf{red}(e', \sigma')
\end{aligned}
$$

The predicate $\mathit{safe}_\varphi(e,\sigma)$ says that executing e never gets stuck, and if it terminates in a value v then the value satisfies the predicate φ.

The following *soundness*[2] theorem connects the wp to *safe*.

Theorem 3 (Soundness). *If* $\vdash \mathsf{wp}\ e\ \{v.\ \ulcorner\varphi(v)\urcorner\}$, *then* $\mathit{safe}_\varphi(e,\sigma)$ *for all* σ.

We do not prove soundness here and instead continue with completeness. Completeness is the converse of soundness, where we go from a safety proof at the meta-level to a proof of a corresponding wp:

Theorem 4 (Completeness). *If* $\mathit{safe}_\varphi(e,\sigma)$ *for all* σ, *then* $\vdash \mathsf{wp}\ e\ \{v.\ \ulcorner\varphi(v)\urcorner\}$.

The goal for the remainder of this section is to prove this theorem.

2.2 Sequential Completeness for Terminating Programs

Before proving Theorem 4, we will warm up by proving a weaker version that makes the additional assumption that e is strongly normalizing under $\rightarrow$ (written $\mathsf{SN}_{\rightarrow}$), *i.e.*, terminating when run from any state. Additionally, as our proof is going to proceed by induction, it will be helpful to generalize the statement to get a better induction hypothesis. Thus, we will prove the following

Theorem 5 (Completeness from Strong Normalization). *If* $\mathit{safe}_\varphi(e,\sigma)$ *and* $\mathsf{SN}_{\rightarrow}(e,\sigma)$, *then* $\ast_{(\ell\leftarrow v)\in\sigma}\ \ell\mapsto v \vdash \mathsf{wp}\ e\ \{v.\ \ulcorner\varphi(v)\urcorner\}$.

In this theorem, we use an iterated separating conjunction to get points-to assertions for all the locations in σ and have to derive the wp. The proof will need two lemmas about *safe*.

Lemma 6. *If* $\mathit{safe}_\varphi(e,\sigma)$, *then either* e *is a value satisfying* φ, *or* (e,σ) *is reducible.*

Lemma 7. *If* $\mathit{safe}_\varphi(e,\sigma)$ *and* $(e,\sigma)\rightarrow(e_2,\sigma_2)$, *then* $\mathit{safe}_\varphi(e_2,\sigma_2)$.

Proof of Theorem 5. The proof is by induction on the assumption that (e,σ) is strongly normalizing. This means that we get to assume the induction hypothesis for all immediate successors (e',σ') of (e,σ). Formally, we have the following:

$$\forall e',\sigma'.\ (e,\sigma)\rightarrow(e',\sigma') \Rightarrow \mathit{safe}_\varphi(e',\sigma') \Rightarrow \mathop{\Huge\ast}_{(\ell\leftarrow v)\in\sigma'} \ell\mapsto v \vdash \mathsf{wp}\ e'\ \{w.\ \ulcorner\varphi(w)\urcorner\} \qquad \text{(IH)}$$

and we are left to prove $\mathit{safe}_\varphi(e,\sigma) \Rightarrow \ast_{(\ell\leftarrow v)\in\sigma}\ \ell\mapsto v \vdash \mathsf{wp}\ e\ \{w.\ \ulcorner\varphi(w)\urcorner\}$ under this induction hypothesis.[3] Now, we just need to reason about (at least) one reduction step of e and then apply the induction hypothesis.

We first apply Lemma 6. If e is a value, we apply WpValue which concludes the proof. In the remaining case, since we know our expression is reducible, we know that $e = K[e_1]$ for some K, e_1 and that $(e_1,\sigma)\rightarrow_{\mathsf{base}}(e_2,\sigma')$ for some e_2,σ'. Thus, using WpBind, we turn our goal into $\mathsf{wp}\ e_1\ \{v.\ \mathsf{wp}\ K[v]\ \{w.\ \ulcorner\varphi(w)\urcorner\}\}$. We continue with a case distinction on the base reduction.

Consider the case BasePure where $e_1 \rightarrow_{\mathsf{pure}} e_2$. We apply WpPure to turn our goal into $\mathsf{wp}\ e_2\ \{v.\ \mathsf{wp}\ K[v]\ \{w.\ \ulcorner\varphi(w)\urcorner\}\}$. Applying WpBind "backwards" turns our goal into $\mathsf{wp}\ K[e_2]\ \{w.\ \ulcorner\varphi(w)\urcorner\}$. Since $(e,\sigma)\rightarrow(K[e_2],\sigma)$, we instantiate our induction hypothesis with $(K[e_2],\sigma)$. By Lemma 7, $(K[e_2],\sigma)$ is safe. Also, the state has not changed so we have the needed points-tos. Thus all premises of the induction hypothesis hold, finishing the case.

The four heap-dependent cases remain. In the BaseLoad case, we get that $e_1 = {!}\,\ell$, and that $\sigma(\ell) = v$. First, we remove the points-to for ℓ from the iterated separating conjunction over σ. Next, we use

[2]In Iris, this theorem is usually called *adequacy* for reasons we discuss later in §10.

[3]As usual with this kind of induction on a well-founded reduction relation, there is no separate base case to consider.

it to instantiate and apply WpLoad; the points-to for ℓ is returned in the postcondition. Finally, we re-establish the iterated separating conjunction for σ and finish the case by applying the induction hypothesis. For BaseStore, we do the same but with WpStore. Of course, the points-to now stores a new value, but σ is altered in lockstep, so we can still re-establish the big separating conjunction and apply the induction hypothesis. For BaseFree, we also remove the points-to for ℓ from the iterated separating conjunction, but WpFree does not return the points-to in the postcondition. Since it is also deleted from σ, we can still establish the iterated separating conjunction for the new state before applying the induction hypothesis.

Only the case BaseAlloc remains. Here, $e_1 = \mathbf{ref}(v)$ and $e_2 = \ell_1$ for some $\ell_1 \notin \mathsf{dom}(\sigma)$. When applying WpAlloc, we get a (fresh) non-deterministic location ℓ_2 along with a points-to, which might be different from ℓ_1. Since we own a points-to for all locations in σ, we use PointsToNe to establish that $\ell_2 \notin \mathsf{dom}(\sigma)$. Since $\ell_2 \notin \mathsf{dom}(\sigma)$ and our allocator is non-deterministic, we have that $(e_1, \sigma) \rightarrow_{\mathsf{base}} (\ell_2, \sigma[\ell_2 \leftarrow v])$ is a valid reduction step. Thus, we apply the induction hypothesis to $(K[\ell_2], \sigma[\ell_2 \leftarrow v])$ and finish the proof. □

Taking a step back, the structure of this proof exploits the fact that either e is already a value, and hence satisfies φ by assumption, or else e is reducible, in which case we perform a case distinction on the reduction rule, using WpBind along the way. After applying a proof rule for the corresponding reduction rule, we use WpBind in the opposite direction to return to something that the induction hypothesis can be applied to. In the following section, we introduce an alternative induction principle that allows us to conduct a similar proof but *without* the strong normalization assumption.

2.3 Sequential Completeness for Partial Programs

Before we attempt to eliminate the strong normalization assumption from our proof, we explain how Iris-based logics typically allow us to reason about non-terminating programs. Consider the program $\Omega \triangleq (\lambda x.\, x\, x)\ (\lambda x.\, x\, x)$. It is easy to see that $\Omega \rightarrow_{\mathsf{pure}} \Omega$. Since our wp is intended to cover partial correctness, we should be able to prove the specification $\mathsf{wp}\ \Omega\ \{v.\ \mathsf{False}\}$. But how?

The core reasoning principle for recursion in typical Iris-based logics is *Löb induction*, which surfaces the underlying step-indexed nature of Iris [62, 66]. In the logic, this is exposed by means of the so-called *later modality* [3, 6, 54], written $\triangleright P$. Intuitively, the assertion $\triangleright P$ says that P is only "approximately" true now, but will be actually true after one step of computation.

The connection between program steps and the $\triangleright$ modality can be seen in the rules for the wp in Figure 2. For example, in the rule WpStore the assumption to the left of the turnstile is $\ell \mapsto v * \triangleright(\ell \mapsto w \twoheadrightarrow \Phi(()))$. The left conjunct requires us to own the points-to for ℓ containing some value v before the store occurs. The right conjunct says that—one step later—after the store has happened, we get back $\ell \mapsto w$ (reflecting the updated value) and have to establish the postcondition.

The later modality commutes with most logical connectives, except for the magic wand and the existential quantifier.[4] There are three key rules for working with the later modality: LaterIntro, LaterMono, and Löb.

$$\textsc{LaterIntro}\quad P \vdash \triangleright P \qquad\qquad \textsc{LaterMono}\quad \frac{P \vdash Q}{\triangleright P \vdash \triangleright Q} \qquad\qquad \textsc{Löb}\quad \frac{\triangleright P \vdash P}{\vdash P}$$

Intuitively, LaterIntro tells us that if something is true now, it remains true one step later. (In particular, this means the wp rules with $\triangleright$ imply the ones without $\triangleright$.) The rule LaterMono is the main elimination rule for the later modality: we can eliminate a later modality in our assumptions if we can remove a later modality from the conclusion as well.

The critical rule for reasoning about loops is Löb induction, which reifies induction on the step index: To prove P, it suffices to prove P under the induction hypothesis $\triangleright P$. This allows a coinductive

[4]It commutes with the existential quantifier if the domain of quantification is non-empty.

style of reasoning: we can use the induction hypothesis $\triangleright P$ only after we made "progress" by taking a step in the program, which lets us apply LaterMono to "unlock" the induction hypothesis.

Example 8. We demonstrate how Löb induction allows us to verify the program Ω:

$$\begin{array}{lll} & \vdash \mathsf{wp}\ \Omega\ \{v.\, \mathsf{False}\} & \\ \Leftarrow & \triangleright \mathsf{wp}\ \Omega\ \{v.\, \mathsf{False}\} \vdash \mathsf{wp}\ \Omega\ \{v.\, \mathsf{False}\} & \text{by Löb} \\ \Leftarrow & \Omega \rightarrow_{\mathsf{pure}} \Omega & \text{by WpPure} \end{array}$$

Here, applying Löb allows us to immediately conclude using WpPure. Standard rules for partial-correctness reasoning about recursive functions or while loops can be derived via Löb induction. The power of Löb induction is that it is not tied to a particular program construct but works for all propositions, and the induction hypothesis can have an arbitrary shape. This is useful for proving our completeness theorem.

Theorem 9 (Completeness of SeqLang)**.** *If* $\mathit{safe}_\varphi(e, \sigma)$, *then* $\mathop{\Huge *}_{(\ell \leftarrow v) \in \sigma} \ell \mapsto v \vdash \mathsf{wp}\ e\ \{v.\, \ulcorner \varphi(v) \urcorner\}$.

The proof proceeds similarly to the proof of Theorem 5, but instead of doing induction on the fact that (e, σ) is strongly normalizing, we do Löb induction.[5] Instead of the requirement to exhibit an explicit step with the $\rightarrow$ relation, the induction hypothesis is now guarded by a later. But this is equivalent, since the later modality is the logic's way of requiring us to take a step! Indeed, the old proof continues to work. In all cases, we apply one of the wp rules which introduces a later modality in the goal. By applying LaterMono, we "unlock" the induction hypothesis and use it to conclude the proof.

3 Completeness for a Concurrent Language

We have seen how to prove completeness for an Iris-based program logic for a sequential language. In this section, we show how that approach generalizes to a concurrent language, ConcLang.

3.1 A Concurrent Separation Logic

We begin by extending the grammar, reduction rules, and program logic with support for concurrency. These additions are outlined in Figure 3. On the language side, we add two new primitives. The first is **fork**, which implements unstructured concurrency. The second is an atomic compare-and-swap operation **CAS**, which allows us to write interesting concurrent programs. Threads are modeled using a thread-pool semantics, defined by the relation $\rightarrow_{\mathsf{tp}}$, which works on configurations ρ now storing a list of threads (expressions) instead of just a single one. We will treat this list as isomorphic to a finite partial function $\mathbb{N} \xrightarrow{\text{fin}} \mathit{Expr}$ mapping n to the n-th element in the list. To create new threads, the relations $\rightarrow$ and $\rightarrow_{\mathsf{base}}$ have a new parameter $\vec{e}_f$, which is a list of forked-off expressions. This list is empty in all rules except TpFork, where it is a singleton list since **fork** creates one new thread.

On the program logic side, the important new rule is WpFork, which allows us to spawn a new thread. Applying this rule lets us reason modularly about concurrent threads by splitting ownership of resources between the current thread and the new thread. For example, if we are trying to establish $\mathsf{wp}\ (\mathbf{fork}\ e_1; e_2)\ \{\Phi\}$, then by applying WpBind and WpFork, we are left with a goal that requires us to prove $\mathsf{wp}\ e_1\ \{v.\, \mathsf{True}\} * \mathsf{wp}\ e_2\ \{\Phi\}$ (ignoring the $\Rrightarrow$ and the mask $\mathcal{E}$ for now). That is, we must prove two *separate* wps, one for each thread. This is the key power of concurrent separation logic! We can verify threads in isolation, as long as we can separate the assumptions needed to verify each.

[5]This requires that we move the assumption $\mathit{safe}_\varphi(e, \sigma)$ into the logic as the assertion $\ulcorner \mathit{safe}_\varphi(e, \sigma) \urcorner$.

$$e \in \mathit{Expr} ::= \cdots \mid \mathbf{fork}\, e \mid \mathbf{CAS}(e_1, e_2, e_3), \quad \vec{e} \in \mathcal{L}(\mathit{Expr}), \quad \rho \in \mathit{Conf} \triangleq \mathcal{L}(\mathit{Expr}) \times \mathit{State}$$

$$P, Q \in \mathit{iProp} ::= \cdots \mid \mathsf{wp}_{\mathcal{E}}\; e\; \{v.\, \Phi(v)\} \mid {\Rrightarrow_{\mathcal{E}}} P \mid \boxed{P}^{\mathcal{N}} \mid \bullet^{\gamma} m \mid k \hookrightarrow^{\gamma} v$$

$$\textsc{TpStep}\quad \frac{(T(n), \sigma) \to (e_2, \sigma, \vec{e}_f)}{(T, \sigma) \to_{\mathsf{tp}} (T[n \leftarrow e_2] \mathbin{+\!\!+} \vec{e}_f, \sigma)} \qquad \textsc{TpFork}\quad (\mathbf{fork}\, e, \sigma) \to_{\mathsf{base}} ((), \sigma, [e])$$

$$\textsc{InvAlloc}\quad P \vdash {\Rrightarrow_{\mathcal{E}}} \boxed{P}^{\mathcal{N}} \qquad \textsc{UpdateIntro}\quad P \vdash {\Rrightarrow_{\mathcal{E}}} P \qquad \textsc{UpdateTrans}\quad {\Rrightarrow_{\mathcal{E}}}{\Rrightarrow_{\mathcal{E}}} P \vdash {\Rrightarrow_{\mathcal{E}}} P \qquad \textsc{UpdateFrame}\quad ({\Rrightarrow_{\mathcal{E}}} P) * Q \vdash {\Rrightarrow_{\mathcal{E}}} (P * Q)$$

$$\textsc{UpdateInv}\quad \frac{\mathcal{N} \subseteq \mathcal{E} \qquad P * Q \vdash {\Rrightarrow_{\mathcal{E} \setminus \mathcal{N}}} P * R}{\boxed{P}^{\mathcal{N}} * Q \vdash {\Rrightarrow_{\mathcal{E}}} R} \qquad \textsc{WpAtomicInv}\quad \frac{\mathcal{N} \subseteq \mathcal{E} \qquad \mathsf{Atomic}\, e \qquad P \vdash \mathsf{wp}_{\mathcal{E} \setminus \mathcal{N}}\; e\; \{v.\, P * \Phi(v)\}}{\boxed{P}^{\mathcal{N}} \vdash \mathsf{wp}_{\mathcal{E}}\; e\; \{\Phi\}}$$

$$\textsc{WpUpdateElim}\quad \frac{P \vdash \mathsf{wp}_{\mathcal{E}}\; e\; \{\Phi\}}{{\Rrightarrow_{\mathcal{E}}} P \vdash \mathsf{wp}_{\mathcal{E}}\; e\; \{\Phi\}} \qquad \textsc{WpWand}\quad \mathsf{wp}_{\mathcal{E}}\; e\; \{\Phi\} * (\forall x.\, \Phi(x) \mathrel{-\!\!*} {\Rrightarrow_{\mathcal{E}}} \Psi(x)) \vdash \mathsf{wp}_{\mathcal{E}}\; e\; \{\Psi\}$$

$$\textsc{WpValue}\quad {\Rrightarrow_{\mathcal{E}}} \Phi(v) \dashv\vdash \mathsf{wp}_{\mathcal{E}}\; v\; \{\Phi\} \qquad \textsc{WpFork}\quad \triangleright {\Rrightarrow_{\mathcal{E}}} (\Phi(()) * \mathsf{wp}_{\top}\; e\; \{v.\, \mathsf{True}\}) \vdash \mathsf{wp}_{\mathcal{E}}\; \mathbf{fork}\, e\; \{\Phi\}$$

Fig. 3. Syntax, Semantics, and Program Logic for ConcLang.

But this separation means that if we have a points-to assertion $\ell \mapsto v$, we can only give it to one of the two threads. That suffices if the two threads access disjoint pieces of state, but if they need to access shared state, then we need other features in the logic: *invariants* and *ghost state*. Readers familiar with these features of Iris can skip to §3.2.

Invariants and Masks. If we want two threads to operate on the same heap cell, we need to somehow give both of them access to it. Since the executions of both threads interleave, we can no longer reason locally in each thread, and instead need to describe the *protocol* under which both threads cooperate. In Iris, this is done with the invariant assertions $\boxed{P}^{\mathcal{N}}$, which says that P is an invariant of program execution (it holds before and after each step), and this invariant has been allocated in namespace $\mathcal{N}$. Invariants are freely duplicable and thus shareable among several threads. There is no exclusive ownership attached to an invariant assertion; all it conveys is *knowledge* that the invariant is maintained. For the purposes of this paper,[6] a thread can use $\boxed{P}^{\mathcal{N}}$ to get access to P for *one atomic step* so long as it shows that P holds again after that step is completed. The *mask* $\mathcal{E}$ on the wp is a set of names of invariants that are currently *enabled* and can be opened; this mechanism is used to prevent a thread from opening the same invariant twice. The process of opening invariants is captured by the rule WpAtomicInv. After opening an invariant, the invariant's namespace is removed from the mask. Additionally, we have the side condition Atomic e which says that e is *atomic* in the sense that it reduces to a value in a single reduction step.

[6]For soundness reasons, invariants only give access to $\triangleright P$. In this paper, our invariants are always *timeless*, so we can eliminate this later modality. See Jung et al. [33] for the details and a definition of timelessness.

Example 10. Let us now verify the following specification about a program which shares a location between two threads.[7]

$$\mathsf{wp}_{\top}\ (\mathbf{let}\ x = \mathbf{ref}(0)\ \mathbf{in}\ \mathbf{fork}\ (x \leftarrow 1); !x)\ \left\{v.\ \ulcorner v \in \{0,1\} \urcorner\right\}$$

The proof proceeds by first allocating the location ℓ_x. Next, we can allocate an invariant using InvAlloc (keep ignoring the $\Rrightarrow$ for now). The invariant we allocate is $\boxed{\exists v \in \{0,1\}.\ \ell_x \mapsto v}^{\mathcal{N}}$ for some namespace $\mathcal{N}$ we choose. The choice does not matter in this example. Since invariant assertions are duplicable, we can pass it to both threads when we apply WpFork. Both threads will now proceed to access the invariant. For the forked-off thread, we use WpAtomicInv and since a store operation is atomic, we get access to the points-to for the duration of the operation. Afterwards, we restore the invariant. This is possible since we store 1, and $1 \in \{0,1\}$. Similarly, when performing the load in the primary thread, we open the invariant to learn $v \in \{0,1\}$ which establishes the postcondition.

What we have just done is created an invariant, shared it between two threads, and opened it around atomic operations in each thread. The choice of invariant assertion was up to us, but we had to ensure that all threads restored the invariant after each atomic step. This is the typical base pattern of concurrent program verification.

Ghost State and Updates. While the invariants we just presented are sufficient for reasoning about simple concurrent programs, it has long been known [56] that for general reasoning about concurrent programs, we need some way to augment the physical state of the program with additional *auxiliary* or *ghost* state. Iris's solution is a general form of *user-defined ghost state*, but for this paper we will only need *ghost map* assertions governed by the following rules.

GhostMapLookup
$$\bullet^{\gamma} m * k \hookrightarrow^{\gamma} v \vdash \ulcorner m(k) = v \urcorner$$

GhostMapAlloc
$$\vdash \Rrightarrow_{\mathcal{E}} \exists \gamma.\ \bullet^{\gamma} \emptyset$$

GhostMapInsert
$$\frac{k \notin \mathsf{dom}(m)}{\bullet^{\gamma} m \vdash \Rrightarrow_{\mathcal{E}} \bullet^{\gamma} m[k \leftarrow v] * k \hookrightarrow^{\gamma} v}$$

GhostMapUpdate
$$\bullet^{\gamma} m * k \hookrightarrow^{\gamma} v \vdash \Rrightarrow_{\mathcal{E}} \bullet^{\gamma} m[k \leftarrow v'] * k \hookrightarrow^{\gamma} v'$$

GhostMapDelete
$$\bullet^{\gamma} m * k \hookrightarrow^{\gamma} v \vdash \Rrightarrow_{\mathcal{E}} \bullet^{\gamma} m \setminus \{k\}$$

Ghost maps work with an arbitrary set of values and a countable set of keys. The ghost points-to $k \hookrightarrow^{\gamma} v$ is very similar to the regular points-to, except that it is tied to a ghost heap instance named γ instead of the regular heap. To avoid confusing ghost points-tos with the (physical) points-tos, we sometimes call the former "ghosts-tos". The *authoritative* state $\bullet^{\gamma} m$ describes the entire ghost heap map m. Updating the ghost heap at key k requires ownership of the authoritative state and of the ghosts-to for k. This update is entirely a logical operation (no physical program operation is involved), which is captured by the *update modality* $\Rrightarrow_{\mathcal{E}}$. Intuitively, $\Rrightarrow_{\mathcal{E}} P$ means that we can update the current ghost state to make P hold. Such updates can be executed in the postcondition of a wp (WpWand), and whenever the goal is itself a wp (WpUpdateElim). The structure of the modality is given by UpdateIntro, UpdateTrans, and UpdateFrame. Together, those rules show that this modality is a strong monad with respect to separating conjunction.

The update also shows up in the invariant allocation rule InvAlloc, since allocating a new invariant also updates ghost state. Additionally, we can use such an update to *peek* into an invariant with UpdateInv. This opens the invariant "for 0 steps," which is occasionally useful when one wants to perform ghost updates using parts of the invariant, or derive pure facts from it.

[7]The program uses let-bindings and sequencing, which are easily encoded using lambdas.

3.2 Completeness for Concurrent Programs

Now that we have extended the logic with concurrency primitives, we revisit the question of soundness and completeness. First, our soundness result changes due to the switch to thread-pool semantics, which requires us to state soundness on the level of configurations:

$$\begin{aligned} \textit{safe-tp}_{\varphi}(\vec{e}, \sigma) &\triangleq \forall \vec{e}\,', \sigma'.\, (\vec{e}, \sigma) \rightarrow^{*}_{\mathsf{tp}} (\vec{e}\,', \sigma') \implies \\ &\qquad \forall n, e''.\ \vec{e}\,'[n] = e'' \Rightarrow (\exists v.\, e'' = v \wedge (n = 0 \Rightarrow \varphi(v))) \vee \mathsf{red}(e'', \sigma') \\ \textit{safe}_{\varphi}(e, \sigma) &\triangleq \textit{safe-tp}_{\varphi}([e], \sigma) \end{aligned}$$

Intuitively, threads are never allowed to be stuck, but only the "main" thread, the first in $\vec{e}\,'$, has to satisfy the postcondition φ on termination.[8] For this modified definition, we have the following analogues of Lemma 6 and Lemma 7.

Lemma 11. *If $\textit{safe-tp}_{\varphi}(\vec{e}, \sigma)$, then $\vec{e}[n]$ is either a value (satisfying φ if $n = 0$), or $(\vec{e}[n], \sigma)$ is reducible.*

Lemma 12. *If $\textit{safe-tp}_{\varphi}(\vec{e}, \sigma)$ and $(\vec{e}, \sigma) \rightarrow_{\mathsf{tp}} (\vec{e}_2, \sigma_2)$, then $\textit{safe-tp}_{\varphi}(\vec{e}_2, \sigma_2)$.*

The soundness theorem establishes safety for a thread pool with e as the initial main thread.

Theorem 13 (Soundness). *If $\vdash \mathsf{wp}_{\top}\ e\ \{v.\, \ulcorner \varphi(v) \urcorner\}$, then $\textit{safe}_{\varphi}(e, \sigma)$ for all σ.*

For the completeness theorem, recall the two main facts that were maintained inductively in our previous sequential proof: (1) the current expression e and state σ satisfies *safe*, and (2) we have points-tos for all of the locations in σ. As we move to the concurrent setting, safety is a property of the entire thread pool, and the points-tos for the locations potentially need to be shared across all threads. To deal with these changes, we first introduce ghost state to track the status of the concurrently running threads. Specifically, we use a ghost map of type $\mathbb{N} \xrightarrow{\text{fin}} \textit{Expr}$ to track the thread pool. This way, even though we reason about one thread at a time when proving a wp, we maintain that the overall system satisfies *safe*. Each thread will own the ghost points-to for its thread identifier. Second, in order to allow threads to share access to the state, we put all the heaps points-tos in a shared invariant, along with the authoritative copy of the thread pool ghost state. This invariant looks as follows.

$$I_{compl} \triangleq \boxed{\exists \vec{e}, \sigma.\, \ulcorner \textit{safe-tp}_{\varphi}(\vec{e}, \sigma) \urcorner * \bullet^{\gamma} \vec{e} * \mathop{\huge *}_{(\ell \leftarrow v) \in \sigma} \ell \mapsto v}^{\mathcal{N}}$$

We can now state the core lemma we are going to prove by Löb induction.

Lemma 14 (Per-thread Completeness). *$I_{compl} * n \hookrightarrow^{\gamma} e \vdash \mathsf{wp}_{\top}\ e\ \{v.\, n \hookrightarrow^{\gamma} v\}$.*

Proof. As in the sequential setting, our proof will eliminate the safety assumption to deduce that e is either a value or reducible; in the latter case, we perform a case analysis on which step it takes. Since the safety assumption is maintained by the invariant, we will need to first open the invariant.

This immediately presents us with an issue: We can only keep the invariant open around the next step if e is atomic, but without opening the invariant we do not know anything about e. Therefore, we do not know whether to apply WpAtomicInv or UpdateInv. It seems like the existing rules do not allow us to open the invariant and only then make the decision of whether to keep it open for the next step or not.

However, it turns out that it is possible to derive a rule where this choice is delayed until after the invariant has been opened. In particular, with this rule we can prove Atomic e after opening,

[8]We think of the overall return value of the concurrent execution as being the value that the first thread returns. This also matches the way forked-off threads only have a postcondition of True in WpFork.

using the assumptions previously kept in the invariant. The statement and proof of this rule is somewhat technical (and it uses the law of the excluded middle in the meta-logic); see Lemma 31 in Appendix A for more details. With this lemma, we can open the invariant and use GhostMapLookup and Lemma 11 to learn that e is a value or reducible. If it is a value, we immediately conclude with WpValue. In the remaining case, we derive $e = K[e_1]$ where e_1 is base-reducible. More specifically, we get that $(e_1, \sigma) \rightarrow_{\textsf{base}} (e', \sigma', \vec{e}_f)$ for some σ, e', σ'.

For *pure steps* $e_1 \rightarrow_{\textsf{pure}} e'$, we have to immediately close the invariant again, since pure steps are in general not atomic. While this forces us to give up all assumptions about the state, we do not actually need them, since pure steps do not care about the state. Using WpBind and WpPure, we are left with proving $\mathsf{wp}_\top\ K[e']\ \{\cdots\}$. We also unlocked the Löb induction hypothesis since the pure step allowed us to eliminate a later. But to apply the induction hypothesis, we would need $n \hookrightarrow^{\gamma} K[e']$ while we currently only have $n \hookrightarrow^{\gamma} K[e_1]$. We can update our ghosts-to by opening the invariant again (for 0 steps) and using GhostMapUpdate. To then close the invariant again, we use Lemma 12 to prove that the new configuration is safe.

For TpFork, we have $e_1 = \textbf{fork}\ e_f$. Here, we also close the invariant and proceed similarly to the previous case. But now we need to prove two wp assertions, one for each thread. To do so, we can again open the invariant to update $n \hookrightarrow^{\gamma} K[\textbf{fork}\ e_f]$ to $K[()]$, but also insert a new entry n' (using GhostMapInsert) to create a new ghosts-to $n' \hookrightarrow^{\gamma} e_f$ for the newly created thread. We close the invariant again. We now use our Löb induction hypothesis *twice*, once for the parent thread and once for the new thread. This is possible because our Löb induction hypothesis holds without any resources and is thus duplicable.[9]

For *atomic expressions* modifying memory, we can keep the invariant open. This lets us obtain ownership of the entire heap (the iterated separating conjunction with all points-tos), so we can proceed like in the sequential case. Handling **CAS** is also straightforward since it is very similar to a load or a store. Once this is done, we close the invariant, which once again requires updating the ghosts-tos like in the other cases. □

This lemma, while having a rather technical precondition, contains the "meat" of the completeness proof. Proving a lemma that is the obvious inverse of soundness is now easy:

Theorem 15 (Completeness of ConcLang)**.** *If* $\mathit{safe}_\varphi(e, \sigma)$ *for all* σ*, then* $\vdash \mathsf{wp}_\top\ e\ \{v.\ \ulcorner\varphi(v)\urcorner\}$.

Proof. We instantiate the precondition with the empty heap $\emptyset$ to get $\mathit{safe\text{-}tp}_\varphi([e], \emptyset)$. Next, we use GhostMapAlloc to get an empty ghost map at γ, and insert $0 \leftarrow e$ into it giving us $0 \hookrightarrow^{\gamma} e$. We allocate the invariant I_{compl} since we have all required resources, including the iterated separating conjunction over the empty set. From Lemma 14, we get $\mathsf{wp}_\top\ e\ \{v.\ 0 \hookrightarrow^{\gamma} v\}$, while we must derive $\mathsf{wp}_\top\ e\ \{v.\ \ulcorner\varphi(v)\urcorner\}$. For this, we use WpWand which also allows us to open invariants with UpdateInv. We assume $0 \hookrightarrow^{\gamma} v$ and must show $\Rrightarrow_\top \ulcorner\varphi(v)\urcorner$, for which we once again open I_{compl}, and then use GhostMapLookup and Lemma 11. Since v is a value and thus not reducible, and since $n = 0$, we get $\varphi(v)$ which finishes the proof. □

3.3 The General Recipe

So far, we considered proving completeness for a *concrete* language. However, Iris provides a default program logic based on a *generic* wp construction, suitable for any language Λ that has a concept of expressions e, values v, state σ and evaluation contexts K. The language Λ must also have contextual operational semantics derived from a base reduction rule $\rightarrow_{\textsf{base}}$, with concurrency handled via thread-pool semantics. This wp comes with a language-independent soundness theorem that factors

[9]Using Iris terminology, the induction hypothesis is *persistent*, which allows us to use it several times.

most of the work of proving soundness for a particular language into a reusable building block. This raises the question: is there a similar language-independent building block for completeness?

To determine what that could look like, we return to the proof of Theorem 15 and split it into a generic and a language-specific part. On the language-specific side, we have the individual cases for each base reduction case, and the fact that we need an iterated separating conjunction over the entire state σ (which can look very different for other languages). To abstract over the iterated separating conjunction, we introduce a predicate $\mathsf{S}_\circ : \Lambda.\mathit{State} \to \mathit{iProp}$ collecting all language-specific resources required to represent the state. To abstract over the base reduction case distinction, we formulate a general condition on base reductions that is sufficient to obtain completeness. Our general completeness theorem then only requires that Λ satisfies this condition.

Theorem 16 (Generic Completeness). *Let Λ be a language satisfying Condition 17 given below. If $\mathit{safe}_\varphi(e,\sigma)$, then $\mathsf{S}_\circ(\sigma) \vdash \mathsf{wp}_\top\, e\, \{v.\, \ulcorner\varphi(v)\urcorner\}$.*

To define this general condition, we introduce two auxiliary definitions:

$$\begin{aligned}
\mathit{AtomicPre}(\mathcal{E}, e, \sigma, \Phi) &\triangleq \forall v', \sigma', \vec{e}_f.\ \ulcorner(e,\sigma) \to (v', \sigma', \vec{e}_f)\urcorner * \mathsf{S}_\circ(\sigma') \Rrightarrow\!\!*_{\mathcal{E}} \\
&\qquad \Phi(v') * \mathop{\Large *}_{e_f \in \vec{e}_f} \mathsf{wp}_\top\, e_f\, \{v.\, \mathsf{True}\} \\
\mathit{NonAtomicPre}(\mathcal{E}, e, \Phi) &\triangleq \forall e', \vec{e}_f.\ \big(\forall \sigma.\, \mathsf{S}_\circ(\sigma) \Rrightarrow\!\!*_{\mathcal{E}} \exists \sigma'.\, \ulcorner(e,\sigma) \to^+ (e', \sigma', \vec{e}_f)\urcorner * \mathsf{S}_\circ(\sigma')\big) \Rrightarrow\!\!*_\top \\
&\qquad \mathsf{wp}_\top\, e'\, \{\Phi\} * \mathop{\Large *}_{e_f \in \vec{e}_f} \mathsf{wp}_\top\, e_f\, \{v.\, \mathsf{True}\}
\end{aligned}$$

Note that we use $P \Rrightarrow\!\!*_{\mathcal{E}} Q$ as an abbreviation for $P \mathrel{-\!\!*} \models\!\!\!\Rrightarrow_{\mathcal{E}} Q$.

Condition 17 (Completeness).

$$\begin{aligned}
\ulcorner\mathsf{base\text{-}red}(e,\sigma)\urcorner * \mathsf{S}_\circ(\sigma) \vdash \models\!\!\!\Rrightarrow_{\mathcal{E}} &\big(\ulcorner\mathsf{Atomic}\, e\urcorner * \forall\Phi.\ \triangleright \mathit{AtomicPre}(\mathcal{E}, e, \sigma, \Phi) \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}\big) \lor \\
&\big(\mathsf{S}_\circ(\sigma) * \forall\Phi.\ \triangleright \mathit{NonAtomicPre}(\mathcal{E}, e, \Phi) \mathrel{-\!\!*} \mathsf{wp}_\top\, e\, \{\Phi\}\big)
\end{aligned}$$

To understand this condition, imagine we are trying to prove it for some language Λ, and ignore the grayed-out parts for now (they are only relevant for reduction steps that fork new threads). We get to assume that e is base-reducible, as well as $\mathsf{S}_\circ(\sigma)$. Our proof can now go ahead by case distinction on base-reducibility. The condition offers us two choices for proceeding.

If e is atomic, we proceed by proving $\mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}$ for some arbitrary Φ, under the precondition that $\triangleright \mathit{AtomicPre}(\mathcal{E}, e, \sigma, \Phi)$. Proving this wp will require a language-specific rule and since e is atomic, this rule should just leave us with $\Phi(v')$ for some v'. The only way to prove $\Phi(v')$ is to use *AtomicPre*. That means we have to prove that (e,σ) actually steps to (v',σ') and that we can update $\mathsf{S}_\circ(\sigma)$ to $\mathsf{S}_\circ(\sigma')$.

If e is not atomic, then we only use the second case.[10] In this case, we must give up $\mathsf{S}_\circ(\sigma)$ since the invariant can not be kept open around a non-atomic expression; this is why our goal is now $\mathsf{wp}_\top\, e\, \{\Phi\}$ with the mask $\top$. Again, the next step is to apply some language-specific proof rule. The idea is that this rule makes progress on the proof by reducing e to e' (think of WpPure). As long as we take at least one step in this reduction, we can apply *NonAtomicPre* to obtain a wp for e'. More specifically, applying *NonAtomicPre* forces us to prove that we can reduce e to e' starting in *any* state σ, potentially changing it to σ'.[11] As part of this, we are given $\mathsf{S}_\circ(\sigma)$ and have to update it to $\mathsf{S}_\circ(\sigma')$. If we can prove such a state-independent reduction from e to e', then in exchange we obtain a wp for e'.

[10]Note that the second case is also usable for atomic expressions, which comes in handy since many technically atomic expressions do not affect the state at all.

[11]In the example we considered above, the state will never change here, but in §6 we will see an example of a language where the non-atomic case is useful for state-changing reductions. Similarly, our examples have so far only taken one step whereas the condition allows taking multiple; this is a straightforward generalization of Lemma 12 and also used in §6.

The grayed-out parts handle fork-based concurrency: If any steps we take fork off new threads $\vec{e}_f$, then we need wp assertions not just for e' (or the postcondition for v') but also the new threads.

Theorem 16 is powerful enough to verify many of our case studies. A similar theorem can be proven for sequential languages; this gives rise to a slightly stronger conclusion (see Appendix B). Some case studies come with a custom definition of wp which makes Theorem 16 formally inapplicable; for these we are able to re-prove a very similar theorem by closely following the pattern outlined so far.

4 Case Study: Completeness of HeapLang

HeapLang is the built-in language of Iris and is often used as a starting point for other Iris-based program logics. HeapLang is very similar to ConcLang, with two main differences. First, freeing a heap location leaves behind a tombstone that prevents this location from being re-allocated in the future. Second, HeapLang has support for prophecy variables [34].

The first point is relevant for completeness because it means a program can assert that two separately allocated locations are different even if the first location is freed before the second one is allocated. HeapLang did not come with a proof rule intended to cover this case, so it may seem like we cannot prove the correctness of such a program and hence the logic is not complete. However, it turns out that the existing rules for HeapLang's "metadata" mechanism (which allows the proof to associate arbitrary ghost data with a physical location) are sufficient to derive correctness of that program. By adding "metadata" tokens in I_{compl}, we are able to finish the completeness proof.

Prophecy variables are technically challenging because of how they are embedded in the operational semantics. For space reasons, we do not explain the full details here. The main take-away is that the general recipe from §3.3 is strong enough to also cover this case. It is worth noting that we slightly adjust the operational semantics to obtain completeness: the operation for "resolving" a prophecy variable does not actually require that the variable exists, but no sensible proof rule can be given for the case where the variable does not exist.

Overall, this case study shows that our approach to proving completeness has the expected effect of identifying examples of programs that are correct with respect to the operational semantics, but cannot be verified with the existing proof rules. With only minor tweaks to the language and the logic, we establish completeness of HeapLang.

5 Case Study: Completeness of a Total Correctness Logic

Apart from the wp e $\{\Phi\}$ assertion for partial correctness, Iris also comes with a variant wp e $[\Phi]$ for total correctness [40]. This new connective satisfies the following soundness theorem.

Theorem 18 (Total Soundness). *If* $\vdash \mathsf{wp}_\top\ e\ \left[v.\ \ulcorner\varphi(v)\urcorner\right]$, *then* $\mathit{safe}_\varphi(e,\sigma) \wedge \mathsf{SN}_{\longrightarrow_{\mathsf{tp}}}([e],\sigma)$ *for all* σ.

This total wp connective is concurrent and proves *scheduler-independent* termination, requiring the program to terminate even under a maximally demonic scheduler.[12] The rules for the total wp are very similar to the ones discussed so far. The only difference is that none of the rules contain any later modalities, so that we are not allowed to eliminate a later modality when taking a step. This makes Löb induction impossible, forcing us to prove that programs actually terminate, typically by induction in the meta-level logic.

This also means that we cannot use Löb induction in our completeness proof either. Instead, we use the strong normalization assumption, echoing §2.2. Our proof mostly follows the general recipe of §3.3, specifically Theorem 16, and can be split into a language-specific total completeness

[12]See [40, p. 11] for a discussion on the limitations of this definition.

condition and a generic part. In fact, the total completeness condition is identical to Condition 17, except that the later modalities are removed.[13]

Theorem 19 (Generic Total Completeness)**.** *Let* Λ *be a language that satisfies the total completeness condition. If* $\mathit{safe}_\varphi(e, \sigma) \land \mathsf{SN}_{\rightarrow_{\mathsf{tp}}}([e], \sigma)$*, then* $\mathsf{S}_\circ(\sigma) \vdash \mathsf{wp}_\mathsf{T}\ e\ \left[v.\ \ulcorner\varphi(v)\urcorner\right]$.

The heart of the proof is a thread-local lemma akin to Lemma 14. The challenge in this proof is that it has to be done by induction on the strong normalization of some configuration $\rho_l = (\vec{e_l}, \sigma_l)$. But since all our knowledge about the "current" configuration is held inside the invariant, we do not have this "current" configuration available to do our induction. It seems like we need to open the invariant *before* starting the induction, but this does not lead to a sufficient induction hypothesis.

The solution is to do induction on a *lower bound* of the current configuration. Since the current configuration is not accessible at the start of the proof, we use additional ghost state to track past configurations that can reach the current configuration, one of which will be ρ_l. The proof is then by strong induction on this lower bound ρ_l being strongly normalizing.

As such, our induction hypothesis holds for all (transitive) successors of ρ_l. Once we open the invariant, we learn (from the additional ghost state) that $\rho_l \rightarrow^*_{\mathsf{tp}} (\vec{e}, \sigma)$, where $(\vec{e}, \sigma)$ is the "current" configuration. As usual in our completeness proofs, we need to take a step $(\vec{e}, \sigma) \rightarrow_{\mathsf{tp}} (\vec{e}', \sigma')$ to a new configuration $(\vec{e}', \sigma')$. But by transitivity, this is also reachable from the lower bound ρ_l, so the induction hypothesis is applicable and we conclude the proof.

We instantiate this general recipe with the total correctness logic for HeapLang.

6 Case Study: Completeness of λ_{Rust}

RustBelt [32] develops a logical relation for a Rust-like type system. As part of this, they define a core calculus called λ_{Rust} that captures the most interesting aspects of Rust, along with a program logic that is used to define the logical relation. We have applied our methodology to their calculus, uncovering some missing proof rules and, more interestingly, showing that the existing proof rules for memory accesses are complete for the non-standard memory model of this language.

λ_{Rust} is superficially similar to ConcLang. The main difference is the memory model, which distinguishes between atomic and non-atomic accesses, and is defined in a way that causes programs with data races to be stuck. We say a data race occurs when there are two concurrent accesses to the same location, in which at least one is a write and at least one is non-atomic. By defining the semantics to make data races get stuck, if we then show that a program is safe (meaning it never gets stuck), then we know it is data race free. To detect data races in the semantics, λ_{Rust} essentially equips every memory location with a reader-writer lock. More precisely, a non-atomic access takes two steps: the first step acquires the lock, getting stuck if this is not possible; this produces an administrative redex. The second step performs the desired operation and releases the lock. Atomic accesses just ensure the lock is not held in a conflicting way; they only take a single step.

The program logic of λ_{Rust} entirely hides the existence of these reader-writer locks. The points-to assertion $\ell \mapsto v$ ensures that the lock is currently not taken. λ_{Rust} also supports fractional permissions [7], with $\ell \mapsto_q v$ for $q < 1$ allowing for the existence of concurrent readers.[14] With this setup, the proof rules for loads and stores are entirely standard. In particular, the rules for atomic and non-atomic loads and stores are *the same*! However, only atomic accesses are Atomic, so WpAtomicInv cannot be used with non-atomic accesses.

[13]See Condition 34 in Appendix C for its exact form.

[14]In fact, all our ghosts-tos and points-tos (in ConcLang and HeapLang) are fractional, but we have so far omitted this since it was not relevant for completeness. In λ_{Rust}, the completeness proof actually has to make use of fractions.

Proving completeness of the data-race-free program logic. The completeness proof uses the general recipe from §3.3. However, we have to adjust I_{compl}: previously, that invariant had exclusive ownership of *the entire heap*. This poses an issue since we cannot open the invariant around non-atomic accesses, preventing access to the points-to we need. But observe that in the specific case of a non-atomic store, there is no reason why the points-to *must* be in the invariant: No other thread will access this location, because there would otherwise be a data race which would contradict the assumption that the program is safe. Similarly, for reads, we know that other threads will not perform writes, so that we should be able to take a fraction of the points-to out of the invariant.

The task is now to construct an invariant that—for each location—tracks which (if any) accesses are currently happening on that location. If an access is ongoing, we temporarily remove the points-to from the invariant and replace it with a proof that no other thread can possibly be performing a concurrent access. (This is inspired by the construction used to reason about optimizations exploiting the absence of data races in Simuliris [17, §3].) Ignoring concurrent reads for now, a simplified version of this invariant for each heap location $\sigma(\ell) = (v, d)$ with value v and reader-writer lock state d is:

$$(\ell \mapsto_1 v \lor (\exists n, e.\, n \hookrightarrow^{\gamma}_{1/2} e * \ulcorner \textit{write-about-to-happen}(e) \urcorner)) * \ulcorner d = \mathsf{Unlocked} \urcorner$$

That is, we either have the full points-to, or we know that thread n is about to perform a non-atomic write. The invariant also asserts that the lock is not held, *i.e.*, there are no memory accesses *currently* going on. This may seem surprising—since the invariant always holds, does that not mean that there can never be an ongoing memory access? Let us take a look at what happens in the proof.

The relevant case in our completeness proof (*i.e.*, in the proof of Condition 17) is the case where a thread is about to perform a non-atomic write to some location ℓ. We open the invariant and prove that only the left disjunct of the per-location invariant can be satisfied: if the right disjunct were active, this would mean another thread were performing a write concurrently with the current thread, which contradicts our assumption that the program is safe. We can therefore switch the invariant to the right disjunct, taking the (physical) points-to out of the invariant while trading in half the thread ghosts-to. Next, we use the *non-atomic* case in Condition 17 (the one that we only used for pure steps so far). This means we have to close the invariant—but crucially, we still own the points-to! We can now take *two* steps in the current thread using the usual proof rule for non-atomic writes. Afterwards, we apply *NonAtomicPre*, which requires us to prove that this reduction is indeed state-independent. This relies on us holding exclusive ownership of ℓ, which implies that no other thread can interfere with us. We know that the per-location invariant is in the right disjunct, so we can switch it to the left disjunct by giving back the points-to and obtaining the second half of our thread ghosts-to. Since we took two steps at once, we skipped over the intermediate state where the reader-writer lock is held, which means that the $d = \mathsf{Unlocked}$ part of the invariant is also maintained. In other words, thanks to the exclusive ownership of ℓ, we do not actually have to show the invariant for every intermediate state. That is how $d = \mathsf{Unlocked}$ can be "always" true.

To also handle non-atomic reads, the real invariant is more complex, requiring more detailed accounting of the fractional parts loaned out to each thread. We do not spell this out here. The overall completeness theorem has the additional assumption that there are no non-atomic accesses *currently happening* in the starting state σ (*i.e.*, all locks are unlocked), which is a reasonable restriction.

Identified completeness gaps. While performing our completeness proofs, we noticed that the existing rules in λ_{Rust} were not complete. The missing rules were unrelated to data races, but instead about the semantics of comparisons, which can be non-deterministic when comparing pointers to already deallocated parts of memory. We presume that the absence of these rules was an oversight, since RustBelt never verified any programs that depended on this non-determinism.

Since λ_{Rust} used a block-based memory model [43], the program logic also exposes *block tokens* tracking the size of each block, which we put into the state invariant $S_\circ$. In doing so, we realized that these block tokens lacked an exclusivity law, which we therefore added.

7 Case Study: Completeness of a Logic for Execution Time Bounds

Mével et al. [51] develop a logic for bounding the running time of programs by embedding time credits [4] into Iris. Here, we consider a completeness proof for such a logic. Similar to their approach, we extend ConcLang with a **tick** operation,[15] where **tick**(e) encodes the idea that executing e incurs a cost. For instance, if we want to bound the number of comparisons that a sorting function performs, we would replace each comparison $e_1 \leq e_2$ in the function with **tick**$(e_1 \leq e_2)$.

For the semantics of **tick**, the state σ of the program is extended with a natural number counter field $\sigma.tc$. Evaluating **tick**(e) first evaluates e until it reaches a value v, then decrements the $\sigma.tc$ counter by one, and returns v. The **tick** operation gets stuck if the $\sigma.tc$ counter is zero. Thus, if $\sigma.tc = m$ and (e, σ) never gets stuck, then e performs no more than m **tick** operations.[16]

To reason about **tick**, the logic adds an assertion $\$(n)$ that represents permission to perform up to n **tick** operations. These assertions are called *time credits* and have the following rules:

$$\textsc{TimecreditSplit} \quad \$(m+n) \dashv\vdash \$(m) * \$(n) \qquad\qquad \textsc{WpTick} \quad \$(1) * \triangleright \Phi(v) \vdash \mathsf{wp}_\top\ \mathbf{tick}(v)\ \{\Phi\}$$

TimecreditSplit allows us to split and join time credits with the separating conjunction. WpTick lets us reason about a **tick** operation by spending 1 credit.

To prove that a program does at most m **tick** operations, we prove a wp starting with an initial budget of $\$(m)$, as expressed in the logic's soundness theorem.

Theorem 20 (Soundness of Time Credits). *If* $\$(m) \vdash \mathsf{wp}_\top\ e\ \left\{v.\ \ulcorner\varphi(v)\urcorner\right\}$, *then* $\mathit{safe}_\varphi(e, \sigma)$ *for all* σ *with* $\sigma.tc = m$.

Because this theorem implies e never gets stuck and $\sigma.tc = m$, we know that e performs at most m **tick** operations. Using the techniques of this paper, it is straightforward to prove a completeness result showing that the above two rules for time credits are all we need:

Theorem 21 (Completeness of Time Credits). *If* $\mathit{safe}_\varphi(e, \sigma)$ *for all* σ *with* $\sigma.tc = m$, *then* $\$(m) \vdash \mathsf{wp}_\top\ e\ \left\{v.\ \ulcorner\varphi(v)\urcorner\right\}$.

Proof Sketch. Apply the generic completeness recipe (Theorem 16), and instantiate $S_\circ(\sigma)$ as $\left(\mathop{\Huge *}_{(\ell \leftarrow v) \in \sigma.h} \ell \mapsto v\right) * \$(\sigma.tc)$. The proof largely follows the proof of Lemma 14, with an additional case for **tick**. For that case, from the safety assumption we know that $\sigma.tc$ is at least 1, so by using TimecreditSplit, we can pull out $\$(1)$, which we spend using WpTick to justify the **tick** step. □

8 Case Study: Completeness of a Logic for Probabilistic Error Bounds

In this section, we turn to Eris [1], an Iris-based separation logic for proving bounds on the probabilities of errors in higher-order sequential probabilistic programs. The idea is that many randomized programs are designed to satisfy a specification with very high probability, and only

[15]Instead of extending ConcLang, Mével et al. [51] actually implement **tick** as a library in ConcLang. However, such an approach can only be complete for programs that use the library "correctly," which is hard to state, let alone reason about.
[16]It is essential to add **tick** around every operation we actually wish to count as incurring a cost. Mével et al. [51] discuss a translation that adds a **tick** to all steps of the language, while other work has instead parameterized the operational semantics with a *cost model* for each transition, but the basic idea is the same.

fail when some rare event occurs. Eris allows one to prove a bound on this probability of failure. Because Eris's specifications cover a probabilistic property, it does not use Iris's default definition of wp, so we cannot directly apply Theorem 16 or Theorem 19 to establish completeness. Nevertheless, as we will see, the general proof approach developed in the previous sections can be used to obtain a completeness result for Eris.

Eris targets a language that extends SeqLang with a probabilistic choice operator **rand** N, which evaluates to a random integer drawn uniformly from the set $\{0, \dots, N\}$. The language removes **free** and makes allocation deterministic, so that the only non-determinism is the probabilistic choice in **rand**. To define a semantics for this language, the authors first define the function $\text{step} : \mathit{Expr} \times \mathit{State} \to \mathit{Expr} \times \mathit{State} \to [0,1]$, where $\text{step}\ (e,\sigma)\ (e',\sigma')$ gives the probability that (e,σ) steps to (e',σ') in a single step. This plays an analogous role to the $\to$ relation in SeqLang. They then inductively define the function $\text{exec}_n : \mathit{Expr} \times \mathit{State} \to \mathit{Val} \to [0,1]$, where $\text{exec}_n\ (e,\sigma)\ v$ gives the probability that (e,σ) will terminate and return v in at most n steps. The partially applied function $\text{exec}_n\ (e,\sigma)$ can be seen as a *sub-distribution* on values, meaning that the sum of the probabilities it assigns to all values is a number $p \le 1$. The gap between p and 1 represents the probability of non-termination after n steps. Finally, $\text{exec}\ (e,\sigma)\ v$ is defined as the limit of exec_n as $n \to \infty$.

Eris comes in two variants, a *partial* version and a *total* version. In this section, we will focus on the total version, though our Rocq development proves a completeness result for the partial version as well. The key feature of Eris is the *error credit* assertion written $↯(\varepsilon)$ with $\varepsilon \in [0,1]$, which represents a "budget" or "permission" to err with probability ε. Just as time credits represent permission to incur a cost and must be spent every time a **tick** occurs, error credits are spent when a specification fails to hold with some probability. By constraining the initial budget, the probability of failure is bounded from above, as formalized in the soundness theorem for total Eris:

Theorem 22 (Soundness of Total Eris). *If* $↯(\varepsilon) \vdash \text{wp}\ e\ \left[v.\ \ulcorner\varphi(v)\urcorner\right]$, *then* $\Pr_{\text{exec}\,(e,\sigma)}[\varphi] \ge 1 - \varepsilon$ *for all* σ.

The conclusion of this statement says that with probability at least $1-\varepsilon$, evaluating (e,σ) will terminate with a value satisfying φ. In other words, the probability that it either diverges, gets stuck, or returns a value that fails to satisfy φ is at most ε, the initial error budget.

We reason about error credits using the following five rules:

ErrCombine
$$↯(\varepsilon_1) * ↯(\varepsilon_2) \dashv\vdash ↯(\varepsilon_1 + \varepsilon_2)$$

ErrWeaken
$$↯(\varepsilon_1) * \ulcorner\varepsilon_2 < \varepsilon_1\urcorner \vdash ↯(\varepsilon_2)$$

Err1
$$↯(1) \vdash \text{False}$$

TwpRandExp
$$\frac{\sum_{i=0}^{N} \dfrac{\mathscr{E}(i)}{N+1} = \varepsilon}{↯(\varepsilon) \vdash \text{wp}\ \textbf{rand}\,N\ [n\,.\ ↯(\mathscr{E}(n))]}$$

TwpThinAir
$$\frac{\forall \varepsilon'.\ \ulcorner\varepsilon' > \varepsilon\urcorner * ↯(\varepsilon') * P \vdash \text{wp}\ e\ [\Phi]}{↯(\varepsilon) * P \vdash \text{wp}\ e\ [\Phi]}$$

ErrCombine allows us to split and join error credits, much like time credits. Using ErrWeaken, we can weaken our credits to a smaller amount. Err1 says that if we have 1 error credit, then we can derive False. The intuition here is that if we look at the statement of soundness, then owning $↯(1)$ means we only have to show our specification holds with probability at least $1-1=0$, which is trivial, so there is nothing to prove. The rule TwpRandExp allows us to re-distribute our credits across the different branches of **rand** N, as long as the resulting error credits have the same *expected value* as the amount we started with. The user instantiates the rule with a function $\mathscr{E} : \{0,\dots,N\} \to [0,1]$, where $\mathscr{E}(i)$ represents how many error credits we will have in the postcondition when the random sample returns i. The idea is that, if an error will occur on some branch based on the outcome of **rand** N, we shift credits from the non-erroring branches to the erroring branches, so that they have $\lightning(1)$ and can apply Err1. Finally, TwpThinAir says that when proving a wp, if we start with ε credits, then it suffices to show that the proof can be carried out with any ε' that is strictly greater than ε. In other words, this rule allows us to conjure up additional error credits "out of thin air," but the additional error might be arbitrarily small. The justification for this rule relies on a continuity property of wp, which we use to take the limit as ε' converges to ε.

Using these rules, it is possible to verify almost-sure termination of programs, *i.e.*, show that they terminate with probability 1. Almost-surely terminating programs are not necessarily strongly normalizing, but the probability of a diverging execution is 0. In general, reasoning about almost-sure termination is challenging, and the research literature has a number of variants of program logics and proof rules for showing almost-sure termination of probabilistic while loops, with recent interest in characterizing the completeness of these proof rules [46, 49].

We will show that Total Eris is *nearly* complete. The failure of completeness has nothing to do with probabilistic reasoning, but rather comes from the fact that the language uses a deterministic allocator, yet the proof rule for allocation is the same as the one we saw in §2, where we non-deterministically get a points-to for the returned location. Thus, instead we prove completeness for programs that do not allocate (but may make use of existing allocated locations):

Theorem 23 (Completeness of Total Eris). *For all configurations (e, σ) not containing any* **ref** *expressions, if* $\mathrm{Pr}_{\mathrm{exec}\,(e,\sigma)}[\varphi] \geq 1 - \varepsilon$, *then* $\lightning(\varepsilon) * \mathop{\Huge *}_{(\ell \leftarrow v) \in \sigma} \ell \mapsto v \vdash \mathsf{wp}\ e\ \left[v.\ \ulcorner\varphi(v)\urcorner\right]$.

To approach the completeness theorem, we start from executions up to a finite number of steps.

Lemma 24. *For all configurations (e, σ) not containing any* **ref** *expressions, if* $\mathrm{Pr}_{\mathrm{exec}_n(e,\sigma)}[\varphi] \geq 1-\varepsilon$, *then* $\lightning(\varepsilon) * \mathop{\Huge *}_{(\ell \leftarrow v) \in \sigma} \ell \mapsto v \vdash \mathsf{wp}\ e\ \left[v.\ \ulcorner\varphi(v)\urcorner\right]$.

Proof. We first weaken the $\lightning(\varepsilon)$ to $\lightning\left(1 - \mathrm{Pr}_{\mathrm{exec}_n(e,\sigma)}[\varphi]\right)$ via ErrWeaken, and then perform induction on n, with e and σ quantified. The structure mirrors the completeness proof in §2, but with one key difference: whereas that proof performed induction on strong normalization (or, for partial programs, used Löb induction), here we induct on the index n of exec_n.

When $n = 0$, either $e = v$ for some $v \in \varphi$, or $\mathrm{Pr}_{\mathrm{exec}_0(e,\sigma)}[\varphi] = 0$. In both cases, the wp can be easily derived. In the inductive step, we need to show:

If $\forall e', \sigma'.\ \lightning\left(1 - \mathrm{Pr}_{\mathrm{exec}_n(e',\sigma')}[\varphi]\right) * \mathop{\Huge *}_{(\ell \leftarrow v) \in \sigma'} \ell \mapsto v \vdash \mathsf{wp}\ e'\ \left[v.\ \ulcorner\varphi(v)\urcorner\right]$ *(the IH),*
then $\lightning\left(1 - \mathrm{Pr}_{\mathrm{exec}_{n+1}(e,\sigma)}[\varphi]\right) * \mathop{\Huge *}_{(\ell \leftarrow v) \in \sigma} \ell \mapsto v \vdash \mathsf{wp}\ e\ \left[v.\ \ulcorner\varphi(v)\urcorner\right]$.

Here, the nontrivial case happens when e is a reducible expression (and hence not a value). As in §2, we decompose e into $K[e_1]$, where e_1 is base-reducible. We then apply WpBind and proceed by case distinction on the base step e_1 takes. For pure reductions and heap-dependent operations (only loads and stores are possible), the proof follows exactly the same pattern as before.

The genuinely new case arises when e_1 is **rand** N. In that case, $e = K[\mathbf{rand}\ N]$ and the configuration (e, σ) steps to $(K[i], \sigma)$ for every $i = 0, \ldots, N$ with equal probability. The failure probability of this expression is given by:

$$1 - \mathrm{Pr}_{\mathrm{exec}_{n+1}(e,\sigma)}[\varphi] = \sum_{i=0}^{N} \frac{1 - \mathrm{Pr}_{\mathrm{exec}_n(K[i],\sigma)}[\varphi]}{N+1}$$

We can therefore apply TwpRandExp with $\mathcal{E}(i) \triangleq 1 - \mathrm{Pr}_{\mathrm{exec}_n(K[i],\sigma)}[\varphi]$. This yields $\lightning(\mathcal{E}(i))$ for each outcome i, with which we can obtain $\mathsf{wp}\ K[i]\ \left[v.\ \ulcorner\varphi(v)\urcorner\right]$ by the induction hypothesis. □

With this lemma, we can prove Theorem 23. We start by applying TwpThinAir, giving us $\lightning(\varepsilon')$ for some $\varepsilon' > \varepsilon$. From this inequality, we have that $\mathrm{Pr}_{\mathrm{exec}\,(e,\sigma)}[\varphi] > 1 - \varepsilon'$. By monotonicity of exec, there exists n such that $\Pr_{\mathrm{exec}_n(e,\sigma)}[\varphi] \geq 1 - \varepsilon'$. It follows by Lemma 24 that $\mathop{\ast}_{(\ell \leftarrow v) \in \sigma} \ell \mapsto v \ast ↯(\varepsilon') \vdash \mathrm{wp}\ e\ [v.\, \varphi(v)]$.

9 Case Study: Completeness of a Relational Logic

So far, we have looked at instances of Iris used for *unary* reasoning, where a specification describes a single program's behavior. However, Iris has also been used for *relational* reasoning, in which specifications relate the behavior of two programs. This kind of reasoning is useful because, among other applications, it can be used to show that a program *refines* a specification program. A common approach is to embed relational reasoning on top of an existing unary program logic by representing the specification program as a form of ghost state, as in the CaReSL logic [66]. With this approach, one introduces assertions to describe the state of this specification program. In this section, we will look at how this approach works for the case of the ConcLang language.

The logic comes with three new assertions. First, we have an assertion $j \Mapsto e$ which states that the thread at index j in the spec program's thread pool is executing expression e, while $\ell \mapsto_s v$ says that in the spec program's heap, location ℓ points to v. Under the hood, both of these are just Iris ghost state, similar to $k \hookrightarrow^{\gamma} v$. Finally, we have an assertion $\mathrm{specCtx}^{\mathcal{N}}$ which represents the invariant that stores the authoritative copies of the spec program's state. This invariant ensures that the spec state can only be updated in a way that reflects the reduction semantics of the spec program. Then one derives rules for executing steps of the spec program by performing Iris ghost state updates. For example, for any $\mathcal{E}$ such that $\mathcal{N} \subseteq \mathcal{E}$, we have:

$$\mathrm{specCtx}^{\mathcal{N}} \ast j \Mapsto K[\ell \leftarrow w] \ast \ell \mapsto_s v \vdash \Rrightarrow_{\mathcal{E}} (j \Mapsto K[()] \ast \ell \mapsto_s w)$$
$$\mathrm{specCtx}^{\mathcal{N}} \ast j \Mapsto K[!\,\ell] \ast \ell \mapsto_s v \vdash \Rrightarrow_{\mathcal{E}} (j \Mapsto K[v] \ast \ell \mapsto_s v)$$
$$\mathrm{specCtx}^{\mathcal{N}} \ast j \Mapsto K[\mathbf{ref}\, v] \vdash \Rrightarrow_{\mathcal{E}} (\exists \ell.\, j \Mapsto K[\ell] \ast \ell \mapsto_s v)$$
$$\mathrm{specCtx}^{\mathcal{N}} \ast j \Mapsto K[\mathbf{fork}\, e] \vdash \Rrightarrow_{\mathcal{E}} (\exists j'.\, j \Mapsto K[()] \ast j' \Mapsto e)$$

The first two rules perform stores and loads, respectively, using the spec program's points-to assertions for the corresponding location. The third rule allocates a new location, creating a new spec points-to for the new location. The fourth rule forks a thread and creates a fresh $j' \Mapsto e$ that represents the forked-off spec thread. Similar rules can be shown for all of the other primitive commands and pure steps of the language.

Then, to prove that an implementation e refines a specification e', one proves a particular wp for e, as captured by the following soundness theorem.

Theorem 25 (Relational Soundness). *If* $\mathrm{specCtx}^{\mathcal{N}} \ast 0 \Mapsto e' \vdash \mathrm{wp}\ e\ \{v.\, \exists v'.\, 0 \Mapsto v' \ast \ulcorner \varphi(v, v') \urcorner\}$, *then*

(1) $(e, \emptyset)$ *is safe, and*
(2) for all v, $\vec{e}$, *and* σ *such that* $([e], \emptyset) \rightarrow^{*}_{\mathbf{tp}} (v :: \vec{e}, \sigma)$ *there exists* v', $\vec{e}\,'$ *and* σ' *such that* $([e'], \emptyset) \rightarrow^{*}_{\mathbf{tp}} (v' :: \vec{e}\,', \sigma')$ *and* $\varphi(v, v')$.

This theorem requires a proof of the implementation e where in the precondition, spec thread 0 executes e', and in the postcondition, the spec thread has reached some value v' that is related to the result v of the implementation. In the conclusion of the theorem, the first part states the usual safety of e that we expect from the wp. The second part captures the relational nature of the logic: it shows that for every execution of e in which the first thread terminates in some value v, there is a corresponding execution of the spec program e' in which its first thread terminates in a value v' that is related to v under φ. In other words, this shows that e *refines* e'. Intuitively, this soundness theorem follows from the fact that, from the way that specCtx and the spec program are defined, the only way we could have gotten $0 \Mapsto v'$ in the postcondition is by constructing such an execution of e'. Because the relational reasoning is constructed by embedding on top of the unary wp, we get to re-use the soundness proof of the underlying wp. As a result, the proof of Theorem 25 is an easy consequence of the underlying soundness of the wp and the definition of the invariant in specCtx.

Now, given that we know the underlying wp is also complete, can we also derive completeness of the relational logic, *i.e.*, a converse to Theorem 25? To show such a converse, we would get as assumptions that (e, σ) is safe and that e refines e' under relation φ, and we would need to derive the entailment. Applying the completeness property of the underlying wp (Theorem 15), we derive a wp about e of the following form:

$$\mathsf{wp}\ e\ \left\{v. \ulcorner \exists v', \vec{e}, \vec{e}\,', \sigma, \sigma'.\ ([e], \varnothing) \rightarrow^*_{\mathsf{tp}} (v :: \vec{e}, \sigma) \wedge ([e'], \varnothing) \rightarrow^*_{\mathsf{tp}} (v' :: \vec{e}\,', \sigma') \wedge \varphi(v, v') \urcorner\right\}$$

The postcondition here has the pure property about the existence of a corresponding execution of e'. To finish the proof, we now "just" need to show from this pure fact that it is possible to update $0 \Mapsto e'$ to $0 \Mapsto v'$. An obvious idea would be to do so by induction on $([e'], \varnothing) \rightarrow^*_{\mathsf{tp}} (v' :: \vec{e}\,', \sigma')$, showing that for each step in this trace, we can do an update on the ghost state.

However, perhaps surprisingly, this approach is blocked by one fundamental issue: the ghost update rule for allocation shown above. The conclusion of this allocation rule says that we just learn that there exists some location at which the allocation has occurred. But because allocation is non-deterministic, this location may not be the same location as the one allocation happens at in our assumed execution about e'!

In some sense, the notion of refinement we are working with is too brittle: we would like to say that the refinement does not depend on the specific locations selected by the allocation. In fact, by changing the definition of the ghost state and the invariant in specCtx, we can derive the following *stronger* version of the soundness theorem:

Theorem 26 (Strong Relational Soundness)**.**
If $\mathsf{specCtx}^{\mathcal{N}} * 0 \Mapsto e' \vdash \mathsf{wp}\ e\ \left\{v. \exists v'.\ 0 \Mapsto v' * \ulcorner \varphi(v, v') \urcorner\right\}$*, then*

(1) $(e, \varnothing)$ *is safe, and*
(2) For any infinite set S *of locations, for all* v*,* $\vec{e}$*, and* σ *such that* $([e], \varnothing) \rightarrow^*_{\mathsf{tp}} (v :: \vec{e}, \sigma)$ *there exists* v'*,* $\vec{e}\,'$ *and* σ' *such that* $([e'], \varnothing) \rightarrow^*_{\mathsf{tp}} (v' :: \vec{e}\,', \sigma')$ *where* $\varphi(v, v')$ *and* $\mathrm{dom}(\sigma') \subseteq S$*.*

In other words, with this version of the theorem, we get to pick an infinite set S of locations,[17] and we get that *all* of the resulting executions of e' only allocate to locations in S. This ensures that the executions of e' cannot depend too much on some particular choice of allocation. With this stronger definition of specCtx, we introduce an additional ghost state assertion $\mathsf{reserve}(S)$ that allows us to *reserve* a set of locations. This assertion is used in the following two rules for $\mathcal{N} \subseteq \mathcal{E}$:

$$\mathsf{specCtx}^{\mathcal{N}} \vdash \Rrightarrow_{\mathcal{E}} \exists S. \ulcorner \mathsf{infinite}(S) \urcorner * \mathsf{reserve}(S)$$

$$\mathsf{specCtx}^{\mathcal{N}} * j \Mapsto K[\mathbf{ref}\, v] * \mathsf{reserve}(S) * \ulcorner \ell \in S \urcorner \vdash \Rrightarrow_{\mathcal{E}} (j \Mapsto K[\ell] * \ell \mapsto_s v * \mathsf{reserve}(S \setminus \{\ell\}))$$

The first rule allows us to obtain, at any point, some infinite set S of reservations. The second rule says that if $\ell \in S$, where S is a set we have reserved, then we can force the allocation to be ℓ, which removes ℓ from the reservation set. With these stronger rules, we can prove a form of completeness that is converse to Theorem 26.

Theorem 27 (Strong Relational Completeness)**.** *If* e *is an expression such that*

[17] More precisely, in our Rocq development, we use a form of constructive infinite sets that allows for splitting an infinite set into two infinite sets without using a choice principle.

(1) $(e, \emptyset)$ *is safe, and*

(2) For any infinite set S *of locations, for all* v, $\vec{e}$, *and* σ *such that* $(e, \emptyset) \rightarrow^*_{\mathsf{tp}} (v :: \vec{e}, \sigma)$ *there exists* v', $\vec{e}\,'$ *and* σ' *such that* $([e'], \emptyset) \rightarrow^*_{\mathsf{tp}} (v' :: \vec{e}\,', \sigma')$ *where* $\varphi(v, v')$ *and* $\mathsf{dom}(\sigma') \subseteq S$.

then $\mathsf{specCtx}^{\mathcal{N}} * 0 \Rrightarrow e' \vdash \mathsf{wp}\; e\; \{v.\, \exists v'.\, 0 \Rrightarrow v' * \ulcorner \varphi(v, v') \urcorner\}$.

The proof follows the sketch we started with above, except that we start by getting $\mathsf{reserve}(S)$ for some infinite set S. From our assumptions, we can then get that there exists an execution of e' that yields a related value where all allocations are in S. Now, because we have $\mathsf{reserve}(S)$, we can perform ghost updates that match this execution. By using the stronger ghost update rule for allocations, we ensure that all of the allocated locations in our ghost state match the ones in the execution we are given.

10 Semantic Conditions for Completeness

In earlier sections, we examined several existing instantiations of Iris and established the completeness of their proof rules. In this section, we turn to the question of how, when coming up with a new instantiation of Iris, one can try to ensure that the resulting logic will be complete. For this purpose, we zoom out from any concrete case study and consider again the default Iris program logic with its language-independent wp.

As mentioned in §3.3, this logic is first parameterized by a notion of a language Λ, which must satisfy some basic structural properties. The next parameter of the framework is the *state interpretation predicate*, $\mathsf{S}_\bullet$, which is a predicate over program states.[18] In Iris, assertions such as $\ell \mapsto v$ do not directly talk about the physical state; they are just ghost state. The role of $\mathsf{S}_\bullet$ is to define a relationship between that ghost state and the program's physical state. Under the hood, $\ell \mapsto v$ is defined using ghosts-tos, and the corresponding authoritative state is held by the state interpretation. This allows the user to define ghost state assertions that track the physical state. Since $\mathsf{S}_\bullet$ often needs to refer to some ghost state name γ, we typically write $\mathsf{S}^\gamma_\bullet$ to indicate this dependency on γ.

The $\mathsf{S}_\bullet$ is used by Iris in the *definition* of the wp. Whereas many program logics *define* the wp so that it is the weakest precondition, in Iris, the wp is instead defined recursively inside the logic.

Definition 28 (wp)**.** The assertion $\mathsf{wp}_{\mathcal{E}}\; e\; \{\Phi\}$ is the fixed point of the following guarded recursive definition [33]

$$\begin{aligned}
(v \in \mathit{Val}) \quad & \mathsf{wp}_{\mathcal{E}}\; v\; \{\Phi\} \triangleq \Rrightarrow_{\mathcal{E}} \Phi(v) \\
(e \notin \mathit{Val}) \quad & \mathsf{wp}_{\mathcal{E}}\; e\; \{\Phi\} \triangleq \forall \sigma.\, \mathsf{S}_\bullet(\sigma) \mathrel{-\!\!*} {}_{\mathcal{E}}\Rrightarrow_{\emptyset} \ulcorner \mathsf{red}(e, \sigma) \urcorner * \\
& \qquad \Big(\forall e', \sigma', \vec{e}_f.\, \ulcorner (e, \sigma) \rightarrow (e', \sigma', \vec{e}_f) \urcorner \mathrel{-\!\!*} \Rrightarrow_{\emptyset} \triangleright\, {}_{\emptyset}\Rrightarrow_{\mathcal{E}} \\
& \qquad \mathsf{S}_\bullet(\sigma') * \mathsf{wp}_{\mathcal{E}}\; e'\; \{\Phi\} * \mathop{\Huge *}_{e_f \in \vec{e}_f} \mathsf{wp}_{\top}\; e_f\; \{\_.\, \mathsf{True}\}\Big)
\end{aligned}$$

The base case of this definition says that if e is already a value v, then the postcondition $\Phi(v)$ must hold immediately (under an update modality $\Rrightarrow_{\mathcal{E}}$). The inductive step consists of two parts. First, given the state interpretation for the current state σ, we must show $\mathsf{red}(e, \sigma)$, *i.e.*, that the current configuration is reducible. Then, for any e' that e may reduce to under σ, we must show the state interpretation holds for the new state σ', and that the wp holds for e' and any forked threads $\vec{e}_f$. This definition uses a mask-changing variant of the update modality, where the update can access some invariants and leave them open (or close some previously open invariants). The

[18]This is somewhat simplified; we focus on what is relevant for this discussion.

various update modalities along the way allow us to open invariants and modify ghost state in the course of proving these facts, while the later modality $\triangleright$ in the definition guards the recursive occurrence of the wp to ensure that the fixed point exists [11, 33]. The later modality here is also the reason why we can strip a later in hypotheses at each step.

On top of this definition, the framework derives various language-generic rules about the wp, such as WpValue, WpWand, and WpAtomicInv. In turn, the user instantiating the framework is responsible for deriving wp rules for each of the language-specific primitives. These proofs require unfolding the definition of the wp and exploiting the user's selected definition of $S_\bullet$.

The framework provides a generic soundness proof for this definition of wp. More precisely, in Iris this property is called *adequacy*, by analogy to the use of that term in denotational semantics, as it shows that the model of the wp actually says something about a program's behavior.

Theorem 29 (Generic Adequacy). *For all states σ, let $S_\circ^\gamma$ be an Iris predicate on states such that*

$$\vdash\ \mathrel{\models\!\!\!\Rrightarrow} \exists \gamma.\ S_\bullet^\gamma(\sigma) * S_\circ^\gamma(\sigma) \tag{StateAlloc}$$

then for any meta-level predicate φ, $(\forall \gamma.\ S_\circ^\gamma(\sigma) \vdash \mathsf{wp}_\top\ e\ \{v.\ \ulcorner \varphi(v) \urcorner\})$ implies $safe_\varphi(e, \sigma)$.

The condition of this theorem requires us to show the state interpretation and state invariant hold for some initial state σ. In turn, if we then derive a wp about e with $S_\circ^\gamma(\sigma)$ as an assumption, we get that (e, σ) satisfies $safe_\varphi$. All of the specific soundness theorems we have looked at for partial correctness have been consequences of this generic adequacy theorem.

A natural question is whether we can show a kind of converse to this theorem under some assumptions about $S_\bullet$ and $S_\circ$. That is, can we show that whenever an expression satisfies *safe*, the wp holds? Such a converse would not exactly be a completeness theorem, because it would not tell us whether a particular collection of *proof rules* suffice for deriving the wp. But such a property is a necessary condition for completeness, since any complete logic would satisfy it. Thus, if our logic fails to satisfy this property, it has no hope of being complete. Moreover, establishing this would show that the wp is in fact the *weakest* precondition, *i.e.*, it is both a necessary and sufficient precondition. That is because any other sufficient precondition would imply *safe*, and therefore imply the wp.

We say that an instance of wp with this property is *requisite*. The following theorem gives sufficient conditions for requisiteness.

Theorem 30 (Generic Requisiteness). *For all γ, let $S_\circ^\gamma$ be an Iris predicate on states such that*

$$\begin{aligned}\forall e, \sigma, \sigma_l.\ \ulcorner \mathsf{red}(e, \sigma_l) \urcorner \mathrel{-\!\!*} S_\bullet^\gamma(\sigma) \mathrel{-\!\!*} S_\circ^\gamma(\sigma_l) \Rrightarrow \ulcorner \mathsf{red}(e, \sigma) \urcorner\ * \\ \left(\forall e', \sigma', \vec{e}_f.\ \ulcorner (e, \sigma) \rightarrow_{\mathsf{tp}} (e', \sigma', \vec{e}_f) \urcorner \Rrightarrow \exists \sigma_l'.\ \ulcorner (e, \sigma_l) \rightarrow_{\mathsf{tp}} (e', \sigma_l', \vec{e}_f) \urcorner * S_\bullet^\gamma(\sigma') * S_\circ^\gamma(\sigma_l')\right)\end{aligned} \tag{StateUpd}$$

then for any meta-level predicate φ, $safe_\varphi(e, \sigma)$ implies $S_\circ^\gamma(\sigma) \vdash \mathsf{wp}_\top\ e\ \{v.\ \ulcorner \varphi(v) \urcorner\}$.

To get some intuition for the condition StateUpd, we think of $S_\bullet$ as the "authoritative" full description of the physical state σ, whereas $S_\circ$ describes some "fragment" or subset of the state σ_l. Then, if e is reducible in the subset of state σ_l for which we have $S_\circ$, and we are given $S_\bullet$ for the full state σ, then we must first be able to show that e is also reducible in σ. Furthermore, for every e' that e can step to from σ, we must update the $S_\bullet$ accordingly, and moreover, we must show that there is a corresponding step to e' from σ_l that forks the same threads. For this step from σ_l, we have to be able to update $S_\circ$.

This captures a sort of locality of the program semantics: ownership of σ_l is enough to characterize how e steps in any larger state that is compatible with ownership of $S_\circ^\gamma(\sigma_l)$.

Proof sketch of Theorem 30. As in the proof of Theorem 15, we first allocate ghost state that will be used to track the status of all concurrent threads that are created. Exactly like before, we create an invariant that stores $S_\circ$, the ghost map for the threads, and the pure fact that the configuration satisfies $safe\text{-}tp_\varphi$.

$$I_{compl} \triangleq \boxed{\exists \vec{e}, \sigma_l.\ \ulcorner safe\text{-}tp_\varphi(\vec{e}, \sigma_l) \urcorner * \bullet^\gamma \vec{e} * S_\circ(\sigma_l)}^{\mathcal{N}}$$

Still using Löb induction, we prove $I_{compl} * n \hookrightarrow^\gamma e \vdash \mathrm{wp}_\top\ e\ \{v.\ n \hookrightarrow^\gamma v\}$, which has the same shape as the entailment in Lemma 14. However, rather than inverting on the base step as we did in that lemma (which we cannot do here as this proof is generic with respect to the language Λ), we instead unfold the definition of the wp. The proof is trivial if e is a value. Otherwise, we open the invariant to show that e can execute for one step and update the thread pool accordingly. We can do these updates by applying StateUpd. We then close the invariant with the updated state and thread pool, and use the induction hypothesis to conclude. □

11 Related Work

Cook's Original Completeness Result. Cook [12] gave the first form of a completeness proof for a variant of Hoare logic. He noted that if one considers a fully syntactic proof system for both the assertion logic and the Hoare triple rules, then when the programming language under consideration is sufficiently expressive, any such proof system must be incomplete, since otherwise it could be used to decide the halting problem. His key insight was that one can side-step this problem and focus on whether there is any incompleteness that can be attributed to a deficiency in the Hoare logic rules, as opposed to the assertion logic rules. To do so, he examines completeness of the Hoare triple rules under the assumption that one has a complete proof system for the assertion logic, and calls this property *relative* completeness. In the case of Iris, we effectively do something similar, since Iris embeds pure assertions from the meta-logic, and adds a proof rule that reflects all implications from the meta-logic as entailments between pure assertions.

Cook next shows that for a sufficiently expressive assertion language, one can define the strongest postcondition sp (the dual of the weakest precondition) as an assertion in the logic. The main challenge lies in the rule for while loops, where one has to express a loop invariant. Once that is done, completeness follows by observing that if $\{P\}\ e\ \{Q\}$ is semantically valid, then $\vdash \{P\}\ e\ \{\mathrm{sp}(e, P)\}$ and $\mathrm{sp}(e, P) \Rightarrow Q$ by construction. Hence, $\mathrm{sp}(e, P) \vdash Q$ by the assumed completeness of the assertion logic, and we have that $\vdash \{P\}\ e\ \{Q\}$ by the rule of consequence. In contrast, for our proofs, rather than *defining* a weakest precondition or strongest postcondition, we effectively end up showing that the wp provided by Iris is in fact the *weakest* precondition, in some sense.

Completeness for Concurrency Logics. Owicki and Gries [56] introduced a proof system for reasoning about concurrent programs. With their approach, to apply the parallel composition rule to $e_1 \mid\mid\mid e_2$, we need not just Hoare *triples* about e_1 and e_2, but rather full Hoare proof outlines that include each intermediate assertion in the Hoare proof between program statements. Then one checks a non-interference property that requires showing each intermediate assertion used in e_1 is *stable* under each of the possible steps of e_2 and vice-versa. Owicki [55] proved the relative completeness of this system. The idea behind her proof is to start by annotating a program with a collection of *auxiliary variables*. These variables are defined so that by inspecting their values during execution, one can determine the full history of the program's execution and interleaving order up to that point. Using these variables, she defines intermediate assertions that are stable by construction, because they can characterize the reachable states of the program. In our proof, the ghosts-tos that track the status of each thread play a similar role to Owicki's auxiliary variables.

This custom ghost state makes it simpler to track the status of all of the concurrent threads, as compared to annotating the program with variables that encode execution ordering.

One drawback of the Owicki-Gries method is that it is non-modular, because the parallel composition rule requires re-inspecting the proofs of the threads and checking them against each other. Rely-guarantee [31] addresses this by using predicates to abstract over the behaviors of threads: we verify each thread under a *rely* assertion that describes what interference could be caused by its environment, while showing that the thread's behavior upholds a *guarantee* assertion. Then, for parallel composition, we simply check for the compatibility of each thread's rely assertion against the other thread's guarantee. Stirling [61] observes that rely-guarantee is complete relative to Owicki-Gries, and since the latter is complete, so too is rely-guarantee. Xu et al. [68] give a direct proof of relative completeness for rely-guarantee, adapting and simplifying the auxiliary variable construction of Owicki. In Iris, rely-guarantee style reasoning is subsumed by the use of invariants and appropriate ghost state.

Most recently, de Boer and Hiep [14] establish relative completeness of a concurrent separation logic by adapting Owicki's technique for proving relative completeness. They define an abstract generalized version of the idea behind Owicki's proof, which can then be instantiated to derive completeness of different logics. They apply this in particular to a CSL that supports dynamic thread creation. As compared to our work, they restrict attention to a first-order language, whereas we have considered higher-order languages with higher-order state. On the other hand, their general theorem would apply to any logic that satisfies their abstract conditions, whereas we have focused on logics built on the Iris framework and its logical mechanisms.

Sequential Separation Logics. Ishtiaq and O'Hearn [30] establish the completeness of a sequential separation logic that uses Hoare triples by showing that the weakest precondition can be formulated as an assertion in the logic. Yang [69] and Yang and O'Hearn [70] also establish completeness results for the original formulation of sequential separation logic.

Haslbeck and Nipkow [27] prove the relative completeness of three program logics for reasoning about time bounds, including a separation logic with time credits. They restrict attention to a language without a dynamic heap or allocation, so as to be able to compare to the other two non-separation logics under consideration. Our results in §7 consider time credits in a language with allocation.

All of the prior completeness results mentioned so far restrict attention to first-order programs. Honda et al. [29] developed a complete program logic for imperative higher-order programs using so-called characteristic formulae that completely capture the behavior of a program. They note that in the first-order case, such formulae are easily constructed, but that in the higher-order case, the challenge is that the description of a higher-order function's behavior must be somehow parameterized by the behavior of its input. Charguéraud [10] adapted characteristic formulae to the setting of higher-order separation logic and established relative completeness in that setting. In our approach, we side-step the question of finding a way to characterize a program's behavior as a formula in the logic by instead representing the threads of the program as ghost state and embedding pure assertions about safety in our invariants.

12 Conclusion

We have introduced a general pattern for completeness proofs of Iris-based program logics. For the family of logics that are based on Iris' shared program logic, we provide reusable lemmas that simplify such proofs for future work. As part of this, we have identified several small gaps in existing logics, indicating that indeed these proofs do the job that one would expect from completeness: ensuring that the proof rules cover everything relevant about every construct in the language.

Interestingly, we did not have to use most of the features that make Iris so expressive: there is no higher-order ghost state, and while we did use invariants, all our invariants are timeless, meaning we did not have a need for general impredicative invariants. The notable exception here is Löb induction, which *does* play an important role in our proofs. Arguably, this is not very surprising. Features like impredicative invariants and higher-order ghost state are motivated by the desire to give powerful, modular specifications to interesting *libraries*, while completeness talks about proving correctness of a whole program in a single, monolithic proof. Library case studies and proofs of completeness are orthogonal means of assessing the expressivity of a logic. Iris has been extensively evaluated for the former; by adding the latter, our work closes a long-standing gap in the work on Iris.

References

[1] Alejandro Aguirre, Philipp G. Haselwarter, Markus de Medeiros, Kwing Hei Li, Simon Oddershede Gregersen, Joseph Tassarotti, and Lars Birkedal. 2024. Error Credits: Resourceful Reasoning about Error Bounds for Higher-Order Probabilistic Programs. *Proc. ACM Program. Lang.* 8, ICFP, Article 246 (Aug. 2024), 33 pages. doi:10.1145/3674635

[2] Clément Allain and Gabriel Scherer. 2026. Zoo: A Framework for the Verification of Concurrent OCaml 5 Programs using Separation Logic. *Proc. ACM Program. Lang.* 10, POPL (2026), 1702–1729. doi:10.1145/3776701

[3] 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 *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007.* 109–122. doi:10.1145/1190216.1190235

[4] Robert Atkey. 2010. Amortised Resource Analysis with Separation Logic. In *ESOP (LNCS, Vol. 6012).* Springer, 85–103. doi:10.1007/978-3-642-11957-6_6

[5] Lars Birkedal and Aleš Bizjak. 2023. Lecture Notes on Iris: Higher-Order Concurrent Separation Logic. https://iris-project.org/tutorial-material.html. https://iris-project.org/tutorial-pdfs/iris-lecture-notes.pdf Aarhus University.

[6] Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. 2012. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *Log. Methods Comput. Sci.* 8, 4 (2012). doi:10.2168/LMCS-8(4:1)2012

[7] John Boyland. 2003. Checking Interference with Fractional Permissions. In *SAS (LNCS, Vol. 2694).* Springer, 55–72. https://doi.org/10.1007/3-540-44898-5_4

[8] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2019. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019.* 243–258. doi:10.1145/3341301.3359632

[9] Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. 2021. GoJournal: a verified, concurrent, crash-safe journaling system. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021.* 423–439. https://www.usenix.org/conference/osdi21/presentation/chajed

[10] Arthur Charguéraud. 2011. Characteristic formulae for the verification of imperative programs. *SIGPLAN Not.* 46, 9 (Sept. 2011), 418–430. doi:10.1145/2034574.2034828

[11] Krzysztof Ciesielski. 2007. On Stefan Banach and some of his results. *Banach Journal of Mathematical Analysis* 1, 1 (2007), 1–10. doi:10.15352/bjma/1240321550

[12] Stephen A. Cook. 1978. Soundness and Completeness of an Axiom System for Program Verification. *SIAM J. Comput.* 7, 1 (1978), 70–90. doi:10.1137/0207005

[13] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2020. RustBelt meets relaxed memory. *Proc. ACM Program. Lang.* 4, POPL (2020), 34:1–34:29. doi:10.1145/3371102

[14] Frank S. de Boer and Hans-Dieter A. Hiep. 2025. Beyond Concurrent Separation Logic: Who is Afraid of Completeness Proofs?. In *Principles of Formal Quantitative Analysis - Essays Dedicated to Christel Baier on the Occasion of Her 60th Birthday (LNCS, Vol. 15760)*, Nathalie Bertrand, Clemens Dubslaff, and Sascha Klüppelholz (Eds.). Springer, 301–319. doi:10.1007/978-3-031-97439-7_15

[15] Paulo Emílio de Vilhena and François Pottier. 2023. A Type System for Effect Handlers and Dynamic Labels. In *Programming Languages and Systems - 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings.* 225–252. doi:10.1007/978-3-031-30044-8_9

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

[17] Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. 2022. Simuliris: a Separation Logic Framework for Verifying Concurrent Program Optimizations. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–31.

[18] Aïna Linn Georges, Benjamin Peters, Laila Elbeheiry, Leo White, Stephen Dolan, Richard A. Eisenberg, Chris Casinghino, François Pottier, and Derek Dreyer. 2025. Data Race Freedom à la Mode. *Proc. ACM Program. Lang.* 9, POPL (2025), 656–686. https://doi.org/10.1145/3704859

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

[20] Simon Oddershede Gregersen, Chaitanya Agarwal, and Joseph Tassarotti. 2025. Logical Relations for Formally Verified Authenticated Data Structures. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security, CCS 2025, Taipei, Taiwan, October 13-17, 2025.* 1394–1408. doi:10.1145/3719027.3744801

[21] Simon Oddershede Gregersen, Alejandro Aguirre, Philipp G. Haselwarter, Joseph Tassarotti, and Lars Birkedal. 2024. Almost-Sure Termination by Guarded Refinement. *Proc. ACM Program. Lang.* 8, ICFP (2024), 203–233. doi:10.1145/3674632

[22] Simon Oddershede Gregersen, Alejandro Aguirre, Philipp G. Haselwarter, Joseph Tassarotti, and Lars Birkedal. 2024. Asynchronous Probabilistic Couplings in Higher-Order Separation Logic. *Proc. ACM Program. Lang.* 8, POPL (2024), 753–784. doi:10.1145/3632868

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

[24] Armaël Guéneau, Johannes Hostert, Simon Spies, Michael Sammler, Lars Birkedal, and Derek Dreyer. 2023. Melocoton: A Program Logic for Verified Interoperability Between OCaml and C. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 716–744. https://doi.org/10.1145/3622823

[25] Philipp G. Haselwarter, Kwing Hei Li, Alejandro Aguirre, Simon Oddershede Gregersen, Joseph Tassarotti, and Lars Birkedal. 2025. Approximate Relational Reasoning for Higher-Order Probabilistic Programs. *Proc. ACM Program. Lang.* 9, POPL (2025), 1196–1226. doi:10.1145/3704877

[26] Philipp G. Haselwarter, Kwing Hei Li, Markus de Medeiros, Simon Oddershede Gregersen, Alejandro Aguirre, Joseph Tassarotti, and Lars Birkedal. 2024. Tachis: Higher-Order Separation Logic with Credits for Expected Costs. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 313 (Oct. 2024), 30 pages. doi:10.1145/3689753

[27] Maximilian P. L. Haslbeck and Tobias Nipkow. 2018. Hoare Logics for Time Bounds - A Study in Meta Theory. In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I (LNCS, Vol. 10805)*, Dirk Beyer and Marieke Huisman (Eds.). Springer, 155–171. doi:10.1007/978-3-319-89960-2_9

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

[29] Kohei Honda, Nobuko Yoshida, and Martin Berger. 2005. An Observationally Complete Program Logic for Imperative Higher-Order Functions. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005), 26-29 June 2005, Chicago, IL, USA, Proceedings.* IEEE Computer Society, 270–279. doi:10.1109/LICS.2005.5

[30] Samin S. Ishtiaq and Peter W. O'Hearn. 2001. BI as an Assertion Language for Mutable Data Structures. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001.* 14–26. doi:10.1145/360204.375719

[31] Cliff B. Jones. 1983. Tentative Steps Toward a Development Method for Interfering Programs. *ACM Trans. Program. Lang. Syst.* 5, 4 (1983), 596–619. doi:10.1145/69575.69577

[32] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–34.

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

[34] Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2019. The future is ours: prophecy variables in separation logic. *Proc. ACM Program. Lang.* 4, POPL, Article 45 (dec 2019), 32 pages. doi:10.1145/3371113

[35] 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*. ACM, 637–650. https://doi.org/10.1145/2676726.2676980

[36] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, Barcelona, Spain, June 19-23, 2017*. 17:1–17:29. doi:10.4230/LIPICS.ECOOP.2017.17

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

[38] 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)*. Springer, 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

[39] 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

[40] Robbert Krebbers, Luko van der Maas, and Enrico Tassi. 2025. Inductive Predicates via Least Fixpoints in Higher-Order Separation Logic. In *16th International Conference on Interactive Theorem Proving (ITP 2025) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 352)*, Yannick Forster and Chantal Keller (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 27:1–27:21. doi:10.4230/LIPIcs.ITP.2025.27

[41] Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. 2020. Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*. 336–365. doi:10.1007/978-3-030-44914-8_13

[42] Peter John Landin. 1964. The Mechanical Evaluation of Expressions. *Comput. J.* (1964).

[43] Xavier Leroy, Andrew Appel, Sandrine Blazy, and Gordon Stewart. 2012. *The CompCert memory model, version 2*. Technical Report RR-7987. Inria. https://hal.inria.fr/hal-00703441

[44] Kwing Hei Li, Alejandro Aguirre, Simon Oddershede Gregersen, Philipp G. Haselwarter, Joseph Tassarotti, and Lars Birkedal. 2025. Modular Reasoning about Error Bounds for Concurrent Probabilistic Programs. *Proc. ACM Program. Lang.* 9, ICFP, Article 245 (Aug. 2025), 30 pages. doi:10.1145/3747514

[45] Martin Hugo Löb. 1955. Solution of a problem of Leon Henkin. *Journal of Symbolic Logic* 20, 2 (1955), 115–118. doi:10.2307/2266895

[46] Rupak Majumdar and V. R. Sathiyanarayana. 2025. Sound and Complete Proof Rules for Probabilistic Termination. *Proc. ACM Program. Lang.* 9, POPL (2025), 1871–1902. doi:10.1145/3704899

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

[48] 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 '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*. 841–856. doi:10.1145/3519939.3523704

[49] Annabelle McIver, Carroll Morgan, Benjamin Lucien Kaminski, and Joost-Pieter Katoen. 2018. A new proof rule for almost-sure termination. *Proc. ACM Program. Lang.* 2, POPL (2018), 33:1–33:28. doi:10.1145/3158121

[50] Glen Mével and Jacques-Henri Jourdan. 2021. Formal verification of a concurrent bounded queue in a weak memory model. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–29. doi:10.1145/3473571

[51] Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2019. Time Credits and Time Receipts in Iris. In *Programming Languages and Systems*, Luís Caires (Ed.). Springer International Publishing, Cham, 3–29. doi:10.1007/978-3-030-17184-1_1

[52] Alexandre Moine, Stephanie Balzer, Alex Xu, and Sam Westrick. 2026. TypeDis: A Type System for Disentanglement. *Proc. ACM Program. Lang.* 10, POPL (2026), 354–383. doi:10.1145/3776655

[53] Alexandre Moine, Arthur Charguéraud, and François Pottier. 2023. A High-Level Separation Logic for Heap Space under Garbage Collection. *Proc. ACM Program. Lang.* 7, POPL (2023), 718–747. doi:10.1145/3571218

[54] Hiroshi Nakano. 2000. A Modality for Recursion. In *15th Annual IEEE Symposium on Logic in Computer Science, Santa Barbara, California, USA, June 26-29, 2000*. 255–266. doi:10.1109/LICS.2000.855774

[55] Susan S. Owicki. 1975. *Axiomatic Proof Techniques for Parallel Programs*. Technical Report. USA.

[56] Susan S. Owicki and David Gries. 1976. An Axiomatic Proof Technique for Parallel Programs I. *Acta Informatica* 6 (1976), 319–340. doi:10.1007/BF00268134

[57] François Pottier, Armaël Guéneau, Jacques-Henri Jourdan, and Glen Mével. 2024. Thunks and Debits in Separation Logic with Time Credits. *Proc. ACM Program. Lang.* 8, POPL (2024), 1482–1508. doi:10.1145/3632892

[58] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 158–174.

[59] Remy Seassau, Irene Yoon, Jean-Marie Madiot, and François Pottier. 2025. Formal Semantics and Program Logics for a Fragment of OCaml. *Proc. ACM Program. Lang.* 9, ICFP (2025), 128–159. doi:10.1145/3747509

[60] Upamanyu Sharma, Ralf Jung, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2023. Grove: a Separation-Logic Library for Verifying Distributed Systems. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*. 113–129. doi:10.1145/3600006.3613172

[61] Colin Stirling. 1988. A Generalization of Owicki-Gries's Hoare Logic for a Concurrent while Language. *Theor. Comput. Sci.* 58 (1988), 347–359. doi:10.1016/0304-3975(88)90033-3

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

[63] The Rocq Team. 2026. The Rocq Prover. https://rocq-prover.org/.

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

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

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

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

[68] Qiwen Xu, Willem P. de Roever, and Jifeng He. 1997. The Rely-Guarantee Method for Verifying Shared Variable Concurrent Programs. *Formal Aspects Comput.* 9, 2 (1997), 149–174. doi:10.1007/BF01211617

[69] Hongseok Yang. 2001. *Local reasoning for stateful programs*. Ph. D. Dissertation. USA. Advisor(s) Reddy, Uday S. AAI3023240.

[70] Hongseok Yang and Peter W. O'Hearn. 2002. A Semantic Basis for Local Reasoning. In *Proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS '02)*. Springer-Verlag, Berlin, Heidelberg, 402–416.

Received 2026-02-19

A The Strong Invariant Opening Lemma

We provide the following lemma which allows us to open invariants and only then decide if we want to keep them open or not, in particular it allows delaying the proof that some expression is atomic until the invariant has been opened.

Lemma 31. *If $\mathcal{N} \subseteq \mathcal{E}$, and if e is reducible under some σ, then it holds that*

$$\begin{aligned}
&\boxed{P}^{\mathcal{N}} * \\
&(P \Rrightarrow\!\!*_{\mathcal{E}\setminus\mathcal{N}} \\
&\quad \Big(\exists K, e'.\ \ulcorner e = K[e'] \urcorner * \ulcorner \mathsf{Atomic}\, e' \urcorner * \big(\exists \sigma'.\ \ulcorner \mathsf{base\text{-}red}\,(e', \sigma') \urcorner\big) * \\
&\qquad \mathsf{wp}_{\mathcal{E}\setminus\mathcal{N}}\, e'\ \big\{v.\, P * \mathsf{wp}_{\mathcal{E}}\, K[v]\ \{\Psi\}\big\}\Big) \lor \\
&\quad P * \mathsf{wp}_{\mathcal{E}}\, e\ \{\Psi\}) \\
&\vdash \mathsf{wp}_{\mathcal{E}}\, e\ \{\Psi\}
\end{aligned}$$

Proof. Since we know that e is reducible, we have $e = K[e']$ and that e' is base-reducible. Case distinction on if e' is atomic:

- If it is, then we apply WpBind followed by WpAtomicInv. We can assume P, specialize the main assumption with it, and also eliminate the update there to expose the big disjunction. We now do a case distinction on it.
  - The first case is easy, as we get K', e'' such that $e = K'[e'']$ and also e'' is base-reducible. Since the base redex is unique[19], we have $K = K'$ and $e'' = e'$. Our goal is then precisely equal to our assumption.
  - In the second case, we frame P into the postcondition then and apply WpBind backwards to conclude.
- If it is not, we use WpUpdateElim and also do a case distinction on our assumption.
  - In the first case, we again get that $e' = e''$ for an atomic e'', which is a contradiction.
  - In the second case, we are done by assumption. □

Note that this lemma is proven using classical reasoning, to decide if e' is atomic. This is usually decidable for a specific language, but we did not want to add this as an extra assumption to our recipe. The lemma can also be proven using the definition of wp (see Definition 28 in §10) without using classical logic, but since our proofs treat wp axiomatically, this would mean that this lemma becomes an additional axiom.

Further, when using this lemma in the proof of Lemma 14 or Theorem 16, we did not have the assumption that e is reducible yet. But in these proofs, it is easily obtained by an additional opening and closing of the invariant at the start.

B Stronger Sequential Versions of Completeness and Requisiteness Theorems

The wp derived by Theorem 16 takes the state invariant $S_\circ(\sigma)$ in the precondition but only gives back a meta-level postcondition $\varphi(v)$. It turns out that if we restrict the expression e to be sequential, we can derive a stronger version of completeness that gives back a $S_\circ$ for the final state.

To keep the result language-agnostic, rather than restrict the syntax of the language (like Theorem 23), we define sequentiality as a meta-level semantic property saying that the thread pool only ever contains 1 thread:

$$seq(e, \sigma) \triangleq \forall \vec{e}\,', \sigma'.\ ([e], \sigma) \rightarrow^*_{\mathbf{tp}} (\vec{e}\,', \sigma') \Rightarrow |\vec{e}\,'| = 1$$

[19]This base redex uniqueness follows from the general laws that must hold for any language used to instantiate Iris's wp.

We also extend the notion of *safe* to account for the final state.

$$\begin{aligned}
\mathit{ssafe\text{-}tp}_\phi(\vec{e},\sigma) &\triangleq \forall\, \vec{e}\,',\sigma'.\ (\vec{e},\sigma) \rightarrow^*_{\mathbf{tp}} (\vec{e}\,',\sigma') \implies \\
&\qquad \forall n, e''.\ \vec{e}\,'[n] = e'' \Rightarrow (\exists v.\ e'' = v \wedge (n = 0 \Rightarrow \phi(v,\sigma'))) \vee \mathsf{red}(e',\sigma') \\
\mathit{ssafe}_\phi(e,\sigma) &\triangleq \mathit{ssafe\text{-}tp}_\phi([e],\sigma)
\end{aligned}$$

The sequential version of completeness is then stated as following.

Theorem 32 (Generic Sequential Completeness). *Let Λ be a language satisfying the Condition 17. If $\mathit{ssafe}_\phi(e,\sigma) \wedge \mathit{seq}(e,\sigma)$, then $\mathsf{S}_\circ(\sigma) \vdash \mathsf{wp}_\top\ e\ \{v.\ \exists\sigma'.\ \mathsf{S}_\circ(\sigma') * \ulcorner\phi(v,\sigma')\urcorner\}$.*

Proof Sketch. Similar to §2.2, but directly do Löb induction on the theorem statement. □

Similarly, we also have a sequential version of requisiteness.

Theorem 33 (Generic Sequential Requisiteness). *For all γ, let $\mathsf{S}_\circ^\gamma$ be an Iris predicate on states satisfying StateUpd, then for any meta-level predicate ϕ, $\mathit{ssafe}_\phi(e,\sigma) \wedge \mathit{seq}(e,\sigma)$ implies $\mathsf{S}_\circ^\gamma(\sigma) \vdash \mathsf{wp}_\top\ e\ \{v.\ \exists\sigma'.\ \mathsf{S}_\circ(\sigma') * \ulcorner\phi(v,\sigma')\urcorner\}$.*

The significance of Theorem 32 and Theorem 33 is that they unlock the reuse of proofs across program logics. With sequential completeness/requisiteness, we can now verify subprograms using external tools and formally embed the resulting safety theorems into Iris.

Note that the sequentiality here is used in an essential way. When the main thread terminates, it must frame a fragment of the final state, σ', into $\mathsf{S}_\circ(\sigma')$ to conclude its postcondition. This prevents other threads from accessing σ' since this point. If we allowed e to fork child threads, then we would have to show child threads can still make progress after the main thread terminates, without accessing σ', which is impossible because safety permits a child thread to rely on σ' to make progress.

C The Exact Form of Generic Total Completeness Theorem

Condition 34 (Total Completeness).

$$\begin{aligned}
\ulcorner\mathsf{base\text{-}red}(e,\sigma)\urcorner * \mathsf{S}_\circ(\sigma) \vdash {\Rrightarrow}_{\mathcal{E}} \big(&\ulcorner\mathsf{Atomic}\ e\urcorner * \forall\Phi.\ \mathit{TAtomicPre}(\mathcal{E},e,\sigma,\Phi) \twoheadrightarrow \mathsf{wp}_{\mathcal{E}}\ e\ [\Phi]\big) \vee \\
&\big(\mathsf{S}_\circ(\sigma) * \forall\Phi.\ \mathit{TNonAtomicPre}(\mathcal{E},e,\Phi) \twoheadrightarrow \mathsf{wp}_\top\ e\ [\Phi]\big)
\end{aligned}$$

where

$$\begin{aligned}
\mathit{TAtomicPre}(\mathcal{E},e,\sigma,\Phi) &\triangleq \forall v',\sigma',\vec{e}_f.\ \ulcorner(e,\sigma) \rightarrow (v',\sigma',\vec{e}_f)\urcorner * \mathsf{S}_\circ(\sigma') \Rrightarrow\!\!*_{\mathcal{E}} \\
&\qquad \Phi(v') * \textstyle\mathop{\Huge *}_{e_f \in \vec{e}_f} \mathsf{wp}_\top\ e_f\ [v.\ \mathsf{True}] \\
\mathit{TNonAtomicPre}(\mathcal{E},e,\Phi) &\triangleq \forall e',\vec{e}_f.\ \big(\forall\sigma.\ \mathsf{S}_\circ(\sigma) \Rrightarrow\!\!*_{\mathcal{E}} \exists\sigma'.\ \ulcorner(e,\sigma) \rightarrow^+ (e',\sigma',\vec{e}_f)\urcorner * \mathsf{S}_\circ(\sigma')\big) \Rrightarrow\!\!*_\top \\
&\qquad \mathsf{wp}_\top\ e'\ [\Phi] * \mathop{\Huge *}_{e_f \in \vec{e}_f} \mathsf{wp}_\top\ e_f\ [v.\ \mathsf{True}]
\end{aligned}$$

Theorem 19 (Generic Total Completeness). *Let Λ be a language that satisfies Condition 34. If $\mathit{safe}_\varphi(e,\sigma) \wedge \mathsf{SN}_{\rightarrow_{\mathbf{tp}}}([e],\sigma)$, then $\mathsf{S}_\circ(\sigma) \vdash \mathsf{wp}_\top\ e\ [v.\ \ulcorner\varphi(v)\urcorner]$.*