TU Delft

Delft University of Technology

Iris from the ground up

A modular foundation for higher-order concurrent separation logic

Jung, Ralf; Krebbers, Robbert; Jourdan, Jacques-Henri; Bizjak, Aleš; Birkedal, Lars; Dreyer, Derek

DOI
10.1017/S0956796818000151

Publication date
2018

Document Version
Accepted author manuscript

Published in
Journal of Functional Programming

Citation (APA)
Jung, R., Krebbers, R., Jourdan, J.-H., Bizjak, A., Birkedal, L., & Dreyer, D. (2018). Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, *28*, 1-73. Article e20. https://doi.org/10.1017/S0956796818000151

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

Under consideration for publication in J. Functional Programming

Iris from the ground up

A modular foundation for higher-order concurrent separation logic

RALF JUNG
MPI-SWS, Germany
(*e-mail:* jung@mpi-sws.org)

ROBBERT KREBBERS
Delft University of Technology, The Netherlands
(*e-mail:* mail@robbertkrebbers.nl)

JACQUES-HENRI JOURDAN
MPI-SWS, Germany
(*e-mail:* jjourdan@mpi-sws.org)

ALEŠ BIZJAK
Aarhus University, Denmark
(*e-mail:* abizjak@cs.au.dk)

LARS BIRKEDAL
Aarhus University, Denmark
(*e-mail:* birkedal@cs.au.dk)

DEREK DREYER
MPI-SWS, Germany
(*e-mail:* dreyer@mpi-sws.org)

Abstract

Iris is a framework for higher-order concurrent separation logic, which has been implemented in the Coq proof assistant and deployed very effectively in a wide variety of verification projects. Iris was designed with the express goal of simplifying and consolidating the foundations of modern separation logics, but it has evolved over time, and the design and semantic foundations of Iris itself have yet to be fully written down and explained together properly in one place. Here, we attempt to fill this gap, presenting a reasonably complete picture of the latest version of Iris (version 3.1), from first principles and in one coherent narrative.

1 Introduction

Iris is a framework for higher-order concurrent separation logic, implemented in the Coq proof assistant, which we and a growing network of collaborators have been developing actively since 2014. It is the only verification tool proposed so far that supports

- foundational machine-checked proofs of
- deep correctness properties for

- fine-grained concurrent programs in
- higher-order imperative languages.

By *foundational machine-checked proofs*, we mean proofs that are performed directly in a proof assistant against the operational semantics of the programming languages under consideration and assuming only the low-level axioms of mathematical logic (as encoded in the type theory of Coq).[1] By *deep correctness properties*, we mean properties that go beyond mere safety, such as contextual refinement or full functional correctness. By *fine-grained concurrent programs*, we mean programs that use low-level atomic synchronization instructions like compare-and-set (CAS) in order to maximize opportunities for parallelism. And by *higher-order imperative languages*, we mean languages like ML and Rust that combine first-class functions, abstract types, and higher-order mutable references.

Moreover, Iris is general in the sense that it is not tied to a particular language semantics and can be used to derive and deploy a range of different formal systems, including but not limited to: logics for atomicity refinement of fine-grained concurrent data structures (Jung *et al.*, 2015), Kripke logical-relations models for relational reasoning in ML-like languages (Krebbers *et al.*, 2017b; Krogh-Jespersen *et al.*, 2017; Timany *et al.*, 2018; Frumin *et al.*, 2018), program logics for relaxed memory models (Kaiser *et al.*, 2017), a program logic for object capability patterns in a JavaScript-like language (Swasey *et al.*, 2017), and a safety proof for a realistic subset of the Rust programming language (Jung *et al.*, 2018).

In this paper, we report on the semantic and logical foundations of Iris, in particular its latest iteration ("Iris 3.1"). Before getting to those foundations and what is so interesting about them, we begin with a bit of historical background on concurrent separation logic.

1.1 A brief history of concurrent separation logic

Around the turn of the millennium, Peter O'Hearn, John Reynolds, Hongseok Yang, and collaborators proposed *separation logic*, a derivative of Hoare logic geared toward reasoning more modularly and scalably about pointer-manipulating programs (O'Hearn *et al.*, 2001; Reynolds, 2002). Inheriting ideas from earlier work on BI—the logic of "bunched implications" (O'Hearn & Pym, 1999; Ishtiaq & O'Hearn, 2001)—separation logic is a "resource" logic, in which propositions denote not only facts about the state of the program, but also *ownership* of resources. The notion of "resource" in the original separation logic is fixed to be a "heap"—*i.e.,* a piece of the global memory represented as a finite partial mapping from memory locations to the values stored there—and the key primitive proposition about heaps is the "points-to" connective $\ell \mapsto v$, asserting ownership of the singleton heap mapping ℓ to v. If we are verifying an expression e which has $\ell \mapsto v$ in its precondition, then we can assume not only that the location ℓ currently points to v, but also that the "right" to update ℓ is "owned" by e. Hence, while verifying e, we need not consider the possibility that another piece of code in the program might "interfere" with e

[1] As in, *e.g.,* Appel *et al.*'s work on *foundational proof-carrying code* (Appel, 2001; Appel & McAllester, 2001) and later on the Verified Software Toolchain (Appel, 2014). This is in contrast to tools like Chalice (Leino *et al.*, 2009), Dafny (Leino, 2010), or Viper (Müller *et al.*, 2016), which have much larger trusted computing bases because they assume the soundness of non-trivial extensions of Hoare logic and do not produce independently checkable proof terms.

by updating ℓ during e's execution. This resistance to interference in turn lets us verify e modularly, that is, without concern for the environment in which e appears.

Although initially separation logic was intended as a logic for sequential programs, it was not long before O'Hearn made the critical observation that separation logic's built-in support for interference resistance could be equally—if not even more—useful for reasoning about *concurrent* programs. In *concurrent separation logic* (CSL) (O'Hearn, 2007; Brookes, 2007), propositions mean much the same as in traditional separation logic, except that they now denote ownership by whichever *thread* is running the code in question. Concretely, this means that if a thread t can assert $\ell \mapsto v$, then t knows that no other thread can read or write ℓ concurrently, so it can completely ignore the other threads and just reason about ℓ as if it were operating in a sequential setting. In the common case where a piece of state is only operated on by one thread at a time, this is a huge win in terms of simplifying verification!

Of course, there is a catch: At some point threads typically have to communicate with one another through some kind of shared state (be it a mutable heap or message-passing channels), and such communication constitutes an unavoidable form of interference. To reason modularly about such interference, the original CSL used a simple form of *resource invariants*, which were tied to a "conditional critical region" construct for synchronization. O'Hearn showed that, just using the standard rules of separation logic plus a simple rule for resource invariants, one could elegantly verify the safety of rather "daring" synchronization patterns, in which ownership of shared resources is transferred subtly between threads in a way that is not syntactically evident from the program text.

After O'Hearn's pioneering paper on CSL (and Brookes' pioneering soundness proof for it), there followed an avalanche of exciting follow-on work, which extended CSL with more sophisticated mechanisms for modular control of interference and which accounted for ownership transfer at a finer granularity (*e.g.,* via atomic compare-and-swap instructions rather than critical sections), thus supporting the verification of even more "daring" concurrent programs (Vafeiadis & Parkinson, 2007; Feng *et al.*, 2007; Feng, 2009; Dodds *et al.*, 2009; Dinsdale-Young *et al.*, 2010b; Fu *et al.*, 2010; Turon *et al.*, 2013; Svendsen & Birkedal, 2014; Nanevski *et al.*, 2014; da Rocha Pinto *et al.*, 2014; Jung *et al.*, 2015). One important conceptual advance in this line of work was the notion of "fictional separation" (Dinsdale-Young *et al.*, 2010a; Dinsdale-Young *et al.*, 2010b)—the idea that even if threads are concurrently manipulating the same *shared* piece of physical state, one can view them as operating more abstractly on *logically disjoint* pieces of the state, and then use separation logic to reason modularly about the abstract pieces. Several more recent logics have also incorporated support for *higher-order quantification* and *impredicative invariants* (Svendsen & Birkedal, 2014; Jung *et al.*, 2015; Jung *et al.*, 2016; Appel, 2014), which are needed if one aims to verify code in languages with semantically cyclic features (such as ML or Rust, which permit mutable references to values of arbitrary type).

This avalanche of work on concurrent separation logic has had a downside, however, which was articulated presciently by Matthew Parkinson in his position paper *The Next 700 Separation Logics* (Parkinson, 2010): "In recent years, separation logic has brought great advances in the world of verification. However, there is a disturbing trend for each new library or concurrency primitive to require a new separation logic." Furthermore, as CSLs become ever more expressive, each one accumulates increasingly baroque and bespoke

proof rules, which are *primitive* in the sense that their soundness is established by direct appeal to the also baroque and bespoke model of the logic. As a result, it is difficult to understand what program specifications in these logics really mean, how they relate to one another, or whether they can be soundly combined in one reasoning framework.

Parkinson argued that what is needed is a general logic for concurrent reasoning, into which a variety of useful specifications can be encoded via the abstraction facilities of the logic. "By finding the right core logic," he wrote, "we can concentrate on the difficult problems." We feel it is high time to renew Parkinson's quest for "the right core logic" of concurrency.

1.2 Iris

Toward this end, we developed **Iris** (Jung *et al.*, 2015), a higher-order concurrent separation logic with the express goal of simplification and consolidation. The key idea of Iris is that even the fanciest of the interference-control mechanisms in recent concurrency logics can be expressed by a combination of two orthogonal (and already well-known) ingredients: *partial commutative monoids* (PCMs) and *invariants*. PCMs enable the user of the logic to roll their own type of fictional (or "logical" or "ghost") state, which is essential to encoding a wide variety of reasoning mechanisms—like permissions, tokens, capabilities, histories, and protocols—which feature in advanced CSLs. Invariants serve to tie that fictional state to the underlying physical state of the program. Using just these two mechanisms, Jung *et al.* (2015) showed how to take complex primitive proof rules from prior logics and *derive them within* Iris, leading to the slogan: "Monoids and invariants are all you need."

Unfortunately, in Iris's initial incarnation (later dubbed "Iris 1.0"), this slogan turned out to be misleading in two ways:

- **Monoids are *not* enough**. There are certain useful varieties of ghost state, such as "named propositions" (see §3.3), in which the structure of the ghost state must be defined mutually recursively with the language of separation-logic propositions. We refer to such ghost state as *higher-order ghost state*, and for encoding higher-order ghost state, it seems we need something more sophisticated than monoids.
- **Iris is not just monoids + invariants.** Although monoids and invariants did indeed constitute the two main conceptual elements of Iris 1.0—and they are arguably "canonical" in their simplicity and universality—the realization of these concepts in the logic involved a number of interacting logical mechanisms, some of which were simple and canonical, others not so much. For example, several mechanisms for controlling ghost state and invariant namespaces were built in as primitive, along with a notion of *mask-changing view shift* (for performing logical updates to resources that may involve "opening" or "closing" invariants) and *weakest preconditions* (for encoding Hoare triples). Moreover, the primitive proof rules for these mechanisms were non-standard, and their semantic model quite involved, making the justification of the primitive rules—not to mention the very *meaning* of Iris's Hoare-style program specifications—very difficult to understand or explain. Indeed, the Iris 1.0 paper (Jung *et al.*, 2015) avoided even attempting to present the formal model of program specifications in any detail at all.

Our subsequent work on Iris 2.0 (Jung *et al.*, 2016) and Iris 3.0 (Krebbers *et al.*, 2017a) has addressed these two points, respectively:

- In Iris 2.0, to support higher-order ghost state, we proposed a generalization of PCMs to what we call *cameras*. Roughly speaking, a camera can be thought of as a kind of "step-indexed PCM", that is, a PCM equipped with a step-indexed notion of equality (Appel & McAllester, 2001; Birkedal *et al.*, 2011) such that the composition operator of the PCM is appropriately "compatible" with the step-indexed equality. For reasons discussed in §4, step-indexing has always been essential to Iris's model of higher-order separation-logic propositions, and by incorporating step-indexed equality into PCMs, cameras enable us to model ghost state that can embed propositions.
- In Iris 3.0, we aimed to simplify the remaining sources of complexity in Iris by taking the Iris story to its (so to speak) logical conclusion: applying the reductionist Iris methodology to Iris itself! Specifically, at the core of Iris 3.0 is a small, resourceful *base logic*, which distills the essence of Iris to what we argue is a bare minimum: it is a higher-order logic extended with the basic connectives of BI (separating conjunction and magic wand), a predicate for resource ownership, and a handful of simple modalities, but it does not bake in any propositions about programs as primitive. It is only this base logic whose soundness must be proven directly against an underlying semantic model (employing the aforementioned cameras). Moreover, using the handful of mechanisms provided by the base logic, Iris 1.0's fancier mechanisms of mask-changing view shifts and weakest preconditions—and their associated proof rules—*can all be derived within the logic*. And by expressing the fancier mechanisms as derived forms, we can now explain the meaning of Iris's program specifications at a much higher level of abstraction than was previously possible.

1.3 Overview of the paper

One unfortunate consequence of the gradual development of Iris across several conference papers is that (1) these papers are not entirely consistent with one another (since the logic has changed over time), and (2) the design and the semantic foundations of Iris have yet to be fully written down and explained together properly in one place.

Here, we attempt to fill this gap by presenting the latest version of Iris (namely, Iris 3.1—we will discuss the minor changes since Iris 3.0 in §8.1) in as unified and self-contained a manner as possible. Our goal is not to convince the reader that Iris is *useful*, although it certainly is—a number of papers (mentioned at the beginning of this introduction) have already demonstrated this, and a more consolidated presentation of the lessons learned from those papers deserves its own journal article. Rather, our goal here is to present a reasonably complete picture of Iris, from first principles and in one coherent narrative.

Toward that end, we begin in §2 and §3 with a tour of the key features of Iris via a simple illustrative example. We continue in §4 by presenting the key algebraic constructions that we need in order to build a semantic model of Iris, including our novel notion of *cameras*. In §5, we describe the Iris 3.1 base logic and show how to give a model of this base logic in terms of the constructions from the previous section. Then, in §6 and §7, we show how to recover the full-blown program logic of Iris by encoding it on top of the Iris base logic.

Finally, in §8 and §9, we conclude with discussion of various technical points and a detailed comparison with related work.

All the results in this paper have been formalized in Coq. Our Coq source, along with further Iris 3.1 documentation, is freely available at the following URL:

http://iris-project.org

2 A tour of Iris

In this section, we will give a brief tour of Iris, motivating and demonstrating the use of its most important features.

Iris is a generic higher-order concurrent separation logic. *Generic* here refers to the fact that the logic is parameterized by the language of program expressions that one wishes to reason about, so the same logic can be used for a wide variety of languages. For the purpose of this section, to make the discussion more concrete, we instantiate Iris with an ML-like language with higher-order store, fork, and compare-and-set (CAS), as given below:

$$
\begin{aligned}
v \in \mathit{Val} &::= () \mid z \mid \mathtt{true} \mid \mathtt{false} \mid \ell \mid \lambda x.e \mid \ldots && (z \in \mathbb{Z})\\
e \in \mathit{Expr} &::= v \mid x \mid e_1(e_2) \mid \mathtt{fork}\ \{e\} \mid \mathtt{assert}(e) \mid \\
&\qquad \mathtt{ref}(e) \mid !e \mid e_1 \leftarrow e_2 \mid \mathtt{CAS}(e, e_1, e_2) \mid \ldots \\
K \in \mathit{Ctx} &::= \bullet \mid K(e) \mid v(K) \mid \mathtt{assert}(K) \mid \mathtt{ref}(K) \mid !K \mid K \leftarrow e \mid v \leftarrow K \mid \\
&\qquad \mathtt{CAS}(K, e_1, e_2) \mid \mathtt{CAS}(v, K, e_2) \mid \mathtt{CAS}(v, v_1, K) \mid \ldots
\end{aligned}
$$

(We omit the usual operations on pairs and sums.)

The logic includes the usual connectives and rules of higher-order separation logic, some of which are shown in the grammar below. (Actually, many of the connectives given in this grammar are defined as *derived forms* in Iris, and this flexibility is an important aspect of the logic, but we leave further discussion of this point until §5-§7.)

$$
\begin{aligned}
P, Q, R ::= {} & \mathsf{True} \mid \mathsf{False} \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \mid \\
& \forall x.\, P \mid \exists x.\, P \mid P * Q \mid \ell \mapsto v \mid \Box P \mid t = u \mid \\
& \boxed{P}^{\mathcal{N}} \mid \boxed{a}^{\gamma} \mid \overline{\mathcal{V}}(a) \mid \{P\}\, e\, \{v.\, Q\}_{\mathcal{E}} \mid P \Rrightarrow_{\mathcal{E}} Q \mid \ldots
\end{aligned}
$$

The Iris proof rules include the usual rules of concurrent separation logic for Hoare triples as given in Figure 1. Note that HOARE-BIND is a generalization of the usual sequencing rule to arbitrary evaluation contexts.

What makes Iris a *higher-order* separation logic is that universal and existential quantifiers can range over any type, including that of propositions and (higher-order) predicates. Furthermore, notice that Hoare triples $\{P\}\, e\, \{v.\, Q\}_{\mathcal{E}}$ are part of the proposition logic (also often called "assertion logic") instead of being a separate entity. As a consequence, triples can be used in the same way as any logical proposition, and in particular, they can be nested to give specifications of higher-order functions. Hoare triples $\{P\}\, e\, \{v.\, Q\}_{\mathcal{E}}$ are moreover annotated with a mask $\mathcal{E}$ to keep track of which invariants are currently in force. We will come back to invariants and masks in §2.2, but for the time being we omit them.

HOARE-FRAME
$$\frac{\{P\}\ e\ \{w.\ Q\}_{\mathcal{E}}}{\{P * R\}\ e\ \{w.\ Q * R\}_{\mathcal{E}}}$$

HOARE-VAL
$$\{\mathsf{True}\}\ v\ \{w.\ w = v\}_{\mathcal{E}}$$

HOARE-BIND
$$\frac{\{P\}\ e\ \{v.\ Q\}_{\mathcal{E}} \qquad \forall v.\ \{Q\}\ K[v]\ \{w.\ R\}_{\mathcal{E}}}{\{P\}\ K[e]\ \{w.\ R\}_{\mathcal{E}}}$$

HOARE-λ
$$\frac{\{P\}\ e[v/x]\ \{w.\ Q\}_{\mathcal{E}}}{\{P\}\ (\lambda x.e)v\ \{w.\ Q\}_{\mathcal{E}}}$$

HOARE-FORK
$$\frac{\{P\}\ e\ \{\mathsf{True}\}}{\{P\}\ \mathtt{fork}\ \{e\}\ \{\mathsf{True}\}_{\mathcal{E}}}$$

HOARE-ASSERT
$$\{\mathsf{True}\}\ \mathtt{assert}(\mathtt{true})\ \{\mathsf{True}\}_{\mathcal{E}}$$

HOARE-ALLOC
$$\{\mathsf{True}\}\ \mathtt{ref}(v)\ \{\ell.\ \ell \mapsto v\}_{\mathcal{E}}$$

HOARE-LOAD
$$\{\ell \mapsto v\}\ !\ell\ \{w.\ w = v * \ell \mapsto v\}_{\mathcal{E}}$$

HOARE-STORE
$$\{\ell \mapsto v\}\ \ell \leftarrow w\ \{\ell \mapsto w\}_{\mathcal{E}}$$

HOARE-CAS-SUC
$$\{\ell \mapsto v\}\ \mathtt{CAS}(\ell, v, w)\ \{b.\ b = \mathtt{true} * \ell \mapsto w\}_{\mathcal{E}}$$

HOARE-CAS-FAIL
$$\frac{v \neq v'}{\{\ell \mapsto v\}\ \mathtt{CAS}(\ell, v', w)\ \{b.\ b = \mathtt{false} * \ell \mapsto v\}_{\mathcal{E}}}$$

Fig. 1. Basic rules for Hoare triples.

Code:

$$\begin{aligned}
&\mathtt{mk_oneshot} \triangleq \lambda_.\ \mathtt{let}\ x = \mathtt{ref}(\mathtt{inl}(0))\ \mathtt{in}\\
&\quad \{\ \mathtt{tryset} = \lambda n.\ \mathtt{CAS}(x, \mathtt{inl}(0), \mathtt{inr}(n)),\\
&\qquad \mathtt{check} = \lambda_.\ \mathtt{let}\ y = !x\ \mathtt{in}\ \lambda_.\\
&\qquad\qquad \mathtt{match}\ y, !x\ \mathtt{with}\\
&\qquad\qquad\ \ \mathtt{inl}(_)\ , _ \quad \Rightarrow ()\\
&\qquad\qquad \mid \mathtt{inr}(n), \mathtt{inl}(_) \ \Rightarrow \mathtt{assert}(\mathtt{false})\\
&\qquad\qquad \mid \mathtt{inr}(n), \mathtt{inr}(m) \Rightarrow \mathtt{assert}(n = m)\\
&\qquad\qquad \mathtt{end}\ \}
\end{aligned}$$

Specification:

$$\{\mathsf{True}\}\ \mathtt{mk_oneshot}()\ \left\{c.\ \forall v.\ \begin{array}{l}\{\mathsf{True}\}\ c.\mathtt{tryset}(v)\ \{w.\ w \in \{\mathtt{true}, \mathtt{false}\}\} *\\ \{\mathsf{True}\}\ c.\mathtt{check}()\ \{f.\ \{\mathsf{True}\}\ f()\ \{\mathsf{True}\}\}\end{array}\right\}$$

Fig. 2. Example code and specification.

A motivating example. We will demonstrate the higher-order aspects of Iris, and some other of its core features, by verifying the safety of the simple higher-order program given in Figure 2. This program is of course rather contrived, but it serves to showcase the core features of Iris.

The function $\mathtt{mk_oneshot}()$ allocates a oneshot location at x and returns a record c with two closures. (Formally, records are syntactic sugar for pairs.) The function $c.\mathtt{tryset}(n)$ tries to set the location x to n, which will fail if the location has already been set. We use $\mathtt{CAS}$ to ensure correctness of the check even if two threads concurrently try to set the location. The function $c.\mathtt{check}()$ records the current state of the location x and then returns a closure which, if the location x has already been initialized, checks that it does not change.

The specification looks a little funny with most pre- and postconditions being True. The reason for this is that all we are aiming to show here is that the code is *safe*, *i.e.,* that

the assertions[2] "succeed", meaning that the branch with `assert(false)` will never get executed, and in the final branch, n will always equal m. In Iris, Hoare triples imply safety, so we do not need to impose any further conditions.

As is common for Hoare triples about functional programs, the postcondition of every Hoare triple has a binder to refer to the return value. We will omit the binder if the result is always unit.

We use nested Hoare triples to express that `mk_oneshot()` returns closures: Since Hoare triples are just propositions, we can put them into the postcondition of `mk_oneshot()` to describe what the client can assume about c. Furthermore, since Iris is a *concurrent* program logic, the specification for `mk_oneshot()` actually permits clients to call c.`tryset`(n) and c.`check`(), as well as the closure f returned by c.`check`(), concurrently from multiple threads and in any combination.

It is worth pointing out that Iris is an *affine* separation logic, which means that it enjoys the weakening rule $P * Q \Rightarrow P$. Intuitively, this rule lets one *throw away* resources; for example, in the postcondition of a Hoare triple one can throw away ownership of unused memory locations. Since Iris is affine, it does not have the Emp connective, which asserts ownership of no resources (O'Hearn *et al.*, 2001); rather, Iris's True connective, which describes ownership of *any* resources, is the identity of separating conjunction (*i.e.,* $P * \mathsf{True} \Leftrightarrow P$). We discuss this design choice further in §9.5.

High-level proof structure. To perform this proof, we need to somehow encode the fact that we are only performing a *oneshot* update to x. To this end, we will allocate a ghost location $\boxed{a}^{\gamma}$ with name γ and value a, which mirrors the current state of x. This may at first sound rather pointless; why should we record a value in the ghost state that is exactly the same as the value in a particular physical location?

The point is that using ghost state lets us choose what kind of *sharing* is possible on the location. For a physical location ℓ, the proposition $\ell \mapsto v$ expresses full ownership of ℓ (and hence the absence of any sharing of it). In contrast, Iris permits us to choose whatever kind of structure and ownership we want for our ghost location γ; in particular, we can define it in such a way that, although the contents of γ mirror the contents of x, we can freely share ownership of γ once it has been initialized (by a call to `tryset`). This in turn will allow the closure returned by `check` to own a piece of γ witnessing its value after initialization. We will then have an *invariant* (see §2.2) tying the value of γ to the value of x, so we *know* which value that closure is going to see when it reads from x, and we know that that value is going to match y.

Another way to describe what is happening is to say that we are applying the idea of *fictional separation* (Dodds *et al.*, 2009): The separation on γ is "fictional" in the sense that multiple threads can own parts of γ and therefore manipulate the same shared variable x, the two being tied together by an invariant.

With this high-level proof structure in mind, we now explain how exactly ownership and sharing of ghost state can be controlled.

[2] Our semantics here is that `assert`(e) gets stuck if e evaluates to `false`. To that end, we have the rule HOARE-ASSERT for `assert(true)`, but no corresponding rule for `assert(false)`.

2.1 Ghost state in Iris: Resource algebras

Iris allows one to use ghost state via the proposition $\boxed{a}^{\gamma}$, which asserts ownership of a piece a of a ghost location γ. The flexibility of Iris stems from the fact that for each ghost location γ, the user of the logic can pick the type M of its value a, instead of this type being fixed in advance by the logic. However, in order to make it possible to use $\boxed{a}^{\gamma}$ in interesting ways, M cannot just be any type, but should have some additional structure, namely:

- It should be possible to compose ownership of different threads. To make this possible, the type M should have an operator $(\cdot)$ for *composition*. The crucial rule of this operation in the logic is $\boxed{a \cdot b}^{\gamma} \Leftrightarrow \boxed{a}^{\gamma} * \boxed{b}^{\gamma}$ (see GHOST-OP in Figure 4).
- Combinations of ownership $\boxed{a}^{\gamma} * \boxed{b}^{\gamma}$ that do not make sense should be ruled out by the logic (*i.e.,* they should entail False). This happens, for example, when multiple threads claim to have ownership of an exclusive resource. To make this possible, the operator $(\cdot)$ should be partial.

Composition of ownership should moreover be *associative and commutative*, to reflect the associative and commutative nature of separating conjunction. For that reason, *partial commutative monoids* (PCMs) have become the canonical structure for representing ghost state in separation logics. In Iris, we are deviating slightly from this, using our own notion of a *resource algebra* (RA), whose definition is in Figure 3. Every PCM is an RA, but not vice versa—and as we will see in our example, the additional flexibility afforded by RAs results in additional logical expressiveness.

There are two key differences between RAs and PCMs:

1. Instead of partiality, RAs use *validity* to rule out invalid combinations of ownership. Specifically, there is a predicate $\overline{\mathcal{V}} : M \to \mathit{Prop}$ identifying *valid* elements. Validity is compatible with the composition operation (RA-VALID-OP).
   We will see in §4.3 that this take on partiality is necessary when defining the structure of *higher-order ghost state*, *i.e.,* ghost state whose structure depends on *iProp*, the type of propositions of the Iris logic.[3]
2. Instead of having a single unit ε that is an identity to *every* element (*i.e.,* that enjoys $\varepsilon \cdot a = a$ for any a), RAs have a partial function $|-|$ that assigns to an element a its *(duplicable) core* $|a|$, as demanded by RA-CORE-ID. We further demand that $|-|$ is idempotent (RA-CORE-IDEM) and monotone (RA-CORE-MONO) with respect to the *extension order*, defined similarly to that for PCMs (RA-INCL).
   An element can have no core, which is indicated by $|a| = \bot$. In order to conveniently deal with partial cores, we use the metavariable $a^?$ to range over elements of $M^? \triangleq M \uplus \{\bot\}$ and lift the composition $(\cdot)$ to $M^?$. As we will see in §3.1, partial cores help us to build interesting composite RAs from smaller primitives.
   In the special case that an RA *does* have a unit ε, we call it a *unital* RA (uRA). From RA-CORE-MONO, it follows that the core of a uRA is a total function, *i.e.,* $|a| \neq \bot$.
   The idea of a duplicable core is not new; we discuss related work in §9.3.

[3] Note the difference between *Prop*, which denotes the type of propositions of the meta-logic (*e.g.,* Coq), and *iProp*, which denotes the type of propositions of Iris.

A *resource algebra* (RA) is a tuple $(M, \overline{\mathcal{V}} : M \to \mathit{Prop}, |-| : M \to M^?, (\cdot) : M \times M \to M)$ satisfying:

$$\forall a, b, c.\ (a \cdot b) \cdot c = a \cdot (b \cdot c) \qquad \text{(RA-ASSOC)}$$
$$\forall a, b.\ a \cdot b = b \cdot a \qquad \text{(RA-COMM)}$$
$$\forall a.\ |a| \in M \Rightarrow |a| \cdot a = a \qquad \text{(RA-CORE-ID)}$$
$$\forall a.\ |a| \in M \Rightarrow ||a|| = |a| \qquad \text{(RA-CORE-IDEM)}$$
$$\forall a, b.\ |a| \in M \land a \preccurlyeq b \Rightarrow |b| \in M \land |a| \preccurlyeq |b| \qquad \text{(RA-CORE-MONO)}$$
$$\forall a, b.\ \overline{\mathcal{V}}(a \cdot b) \Rightarrow \overline{\mathcal{V}}(a) \qquad \text{(RA-VALID-OP)}$$

where $M^? \triangleq M \uplus \{\bot\}$ with $a^? \cdot \bot \triangleq \bot \cdot a^? \triangleq a^?$

$$a \preccurlyeq b \triangleq \exists c \in M.\ b = a \cdot c \qquad \text{(RA-INCL)}$$
$$a \rightsquigarrow B \triangleq \forall c^? \in M^?.\ \overline{\mathcal{V}}(a \cdot c^?) \Rightarrow \exists b \in B.\ \overline{\mathcal{V}}(b \cdot c^?)$$
$$a \rightsquigarrow b \triangleq a \rightsquigarrow \{b\}$$

A *unital resource algebra* (uRA) is a resource algebra M with an element ε satisfying:

$$\overline{\mathcal{V}}(\varepsilon) \qquad \forall a \in M.\ \varepsilon \cdot a = a \qquad |\varepsilon| = \varepsilon$$

Fig. 3. Resource algebras.

A resource algebra for our example. We will now define the RA that can be used to verify our example, which we call the *oneshot* RA.[4] The goal of this RA is to appropriately reflect the state of the physical location x. The carrier is defined using a datatype-like notation as follows:[5]

$$M \triangleq \mathsf{pending} \mid \mathsf{shot}(n : \mathbb{Z}) \mid \lightning$$

The two important states of the ghost location are: pending, to represent the fact that the single update has not yet happened, and $\mathsf{shot}(n)$, saying that the location has been set to n. We need an additional element $\lightning$ to account for partiality; it is the only invalid element:

$$\overline{\mathcal{V}}(a) \triangleq a \neq \lightning$$

The most interesting part of an RA is, of course, its composition: What happens when ownership of two threads is combined? (Compositions not defined by the following equation are mapped to $\lightning$.)

$$\mathsf{shot}(n) \cdot \mathsf{shot}(m) \triangleq \begin{cases} \mathsf{shot}(n) & \text{if } n = m \\ \lightning & \text{otherwise} \end{cases}$$

[4] In this section, we give an explicit definition of the oneshot RA. This may feel a bit verbose, and indeed, in §3.1, we show that this RA can in fact be defined using a couple of combinators.

[5] Notations for invalid elements have not been entirely consistent among earlier Iris papers. In this paper we use $\lightning$ to denote the invalid element of an RA (if the RA has such an element), $\bot$ to denote the absence of a core, and ε to denote the global unit (if it exists).

This definition has three important properties:

$$\overline{\mathcal{V}}(\mathsf{pending} \cdot a) \quad \Rightarrow \quad \mathsf{False} \qquad \text{(PENDING-EXCL)}$$

$$\overline{\mathcal{V}}(\mathsf{shot}(n) \cdot \mathsf{shot}(m)) \quad \Rightarrow \quad n = m \qquad \text{(SHOT-AGREE)}$$

$$\mathsf{shot}(n) \cdot \mathsf{shot}(n) = \mathsf{shot}(n) \qquad \text{(SHOT-IDEM)}$$

The property PENDING-EXCL says that composition of pending with anything else is invalid. As a result of this, if we *own* pending, we know that no other thread can own another part of this location. Furthermore, SHOT-AGREE says that composition of two $\mathsf{shot}(-)$ elements is valid only if the parameters (*i.e.,* the values picked for the oneshot) are the same. This reflects the idea that once a value has been picked, it becomes the only possible value of the location; every thread agrees on what that value is.

Finally, SHOT-IDEM says that we can also duplicate ownership of the location as much as we want, *once it has been set to some n*. This allows us to share ownership of this piece of ghost state among any number of threads.

Notice that we use the term "ownership" in a rather loose sense here: any element of an RA can be "owned". For elements like $\mathsf{shot}(n)$ that satisfy the property $a = a \cdot a$, owning a is equivalent to owning multiple copies of a—in this particular case, ownership is no longer exclusive, and it may be more appropriate to call this *knowledge*. We can hence think of $\mathsf{shot}(n)$ as representing the knowledge that the value of the location has been set to n. Hence, resource algebras use the same mechanism to serve a double purpose: modeling both (1) ownership of resources and (2) sharing of knowledge.

Coming back to our oneshot RA, we still have to define the core $|-|$:

$$|\mathsf{pending}| \triangleq \bot \qquad |\mathsf{shot}(n)| \triangleq \mathsf{shot}(n) \qquad |\lightning| \triangleq \lightning$$

Note that, since ownership of pending is exclusive, it has no suitable unit element, so we assign no core.

This completes the definition of the oneshot RA. It is now straightforward to verify that this RA satisfies the RA axioms.

Frame-preserving updates. So far, we have defined which states our ghost location can be in and how the state of the location can be distributed across multiple threads. What is still missing, however, is a way of *changing* the ghost location's state. When the ghost state changes, it is important that it remains valid—Iris always maintains the invariant that the state obtained by composing the contributions of all threads is a valid RA element. We call state changes that maintain this invariant *frame-preserving updates*.

The simplest form of frame-preserving update is *deterministic*. We can do a frame-preserving update from a to b (written $a \rightsquigarrow b$) when the following condition is met:

$$\forall c^? \in M^?.\ \overline{\mathcal{V}}(a \cdot c^?) \Rightarrow \overline{\mathcal{V}}(b \cdot c^?)$$

In other words, for any resource (called a *frame*) $c^? \in M^?$ such that a is compatible with $c^?$ (*i.e.,* $\overline{\mathcal{V}}(a \cdot c^?)$), it has to be the case that b is also compatible with $c^?$.

For example, with our oneshot RA it is possible to pick a value if it is still pending:

$$\mathsf{pending} \rightsquigarrow \mathsf{shot}(n) \qquad \text{(ONESHOT-SHOOT)}$$

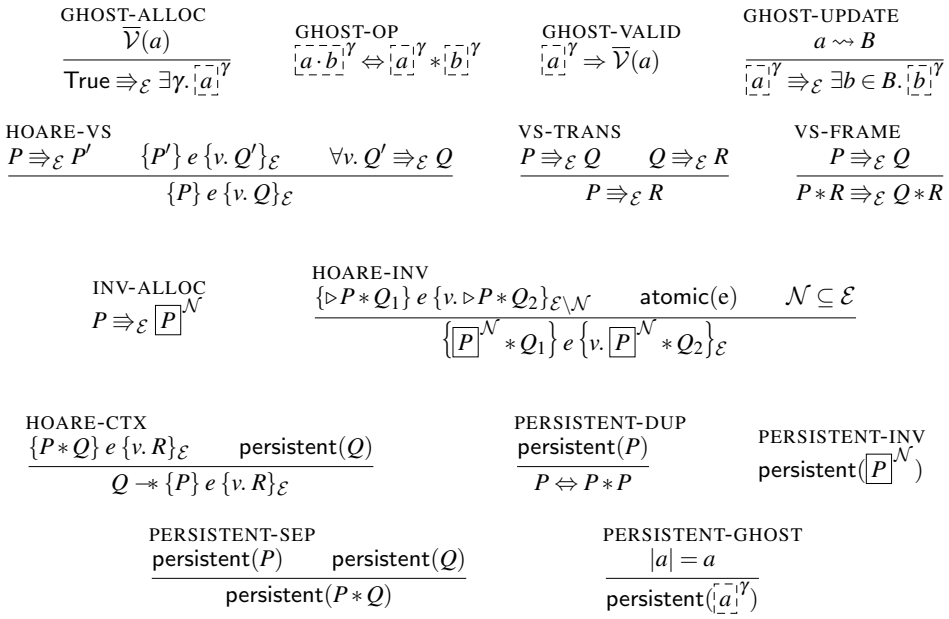

Fig. 4. Some Iris proof rules.

The reason for this is PENDING-EXCL: pending actually is not compatible with *any* element; composition always yields $\lightning$. It is thus the case that from $\overline{\mathcal{V}}(\mathsf{pending} \cdot c^?)$, we know $c^? = \bot$. This makes the rest of the proof trivial.

If we think of the frame $c^?$ as being the composition of the resources owned by all the other threads, then a frame-preserving update is guaranteed not to invalidate the resources of concurrently running threads. The frame can be $\bot$ if no other thread has any ownership of this ghost location.[6] By doing only frame-preserving updates, we know we will never "step on anybody else's toes".

In general, we also permit *non-deterministic* frame-preserving updates (written $a \rightsquigarrow B$) where the target element b is not fixed a priori, but instead a set B is fixed and some element $b \in B$ is picked depending on the current frame. This is formally defined in Figure 3. We will discuss non-deterministic frame-preserving updates further when we encounter our first example of such an update in §3.2.

Proof rules for ghost state. Resource algebras are embedded into the logic using the proposition $\boxed{a}^{\gamma}$, which asserts ownership of a piece a of the ghost location γ. The main connective for manipulating these ghost assertions is called a *view shift* (or *ghost move*): $P \Rrightarrow_{\mathcal{E}} Q$ says that, given resources satisfying P, we can change the ghost state and end up with resources satisfying Q. (We will come back to the *mask* annotation $\mathcal{E}$ in §2.2.) Intuitively, view shifts are like Hoare triples, but without any code—there is just a precondition

[6] Note that the notion of frame-preserving updates is defined for RAs in general, and not just uRAs. To that end, we cannot rely on the presence of a global unit ε to account for the absence of a frame (or the absence of another thread with ownership of the ghost location).

and a postcondition. They do not need any code because they only touch the ghost state, which does not correspond to any operation in the actual program.

The proof rule GHOST-ALLOC in Figure 4 can be used to allocate a new ghost location, with an arbitrary initial state a so long as a is valid according to the chosen RA. The rule GHOST-UPDATE says that we can perform frame-preserving updates on ghost locations, as described above.

All the usual structure rules for Hoare triples also hold for view shifts, like framing (VS-FRAME). The rule HOARE-VS illustrates how view shifts are used in program verification: we can apply view shifts in the pre- and postconditions of Hoare triples. This corresponds to composing the "triples for ghost moves" (*i.e.,* view shifts) with a Hoare triple for e. Doing so does not change the expression in the triple because the ghost state actions performed by the view shifts do not speak about any actual code.

The rule GHOST-OP says that ghost state can be separated (in the sense of separation logic) following the composition operation $(\cdot)$ defined for the RA, and GHOST-VALID encodes the fact that only valid RA elements can ever be owned.

Notice that view shifts ($\Rrightarrow$) are very different from implications ($\Rightarrow$) and wands ($-\!\!*$): the implication $P \Rightarrow Q$ says that whenever P holds, Q necessarily holds too. In contrast, the view shift $P \Rrightarrow Q$ says that whenever P holds, Q holds under the proviso that we change the ghost state. Hence, unlike implication and wand, the only way to eliminate a view shift is through the rule HOARE-VS.

2.2 Invariants

Now that we have set up the structure of our ghost location γ, we have to connect the state of γ to the actual physical value of x. This is done using an *invariant*.

An invariant (Ashcroft, 1975) is a property that holds at all times: each thread accessing the state may assume the invariant holds before each step of its computation, but it must also ensure that it continues to hold after each step. Since we work in a separation logic, the invariant does not just "hold"; it expresses ownership of some resources, and threads accessing the invariant get access to those resources. The rule HOARE-INV realizes this idea as follows:

$$\frac{\{\triangleright P * Q_1\}\, e\, \{v.\, \triangleright P * Q_2\}_{\mathcal{E}\setminus\mathcal{N}} \qquad \mathsf{atomic}(e) \qquad \mathcal{N} \subseteq \mathcal{E}}{\left\{\boxed{P}^{\mathcal{N}} * Q_1\right\}\, e\, \left\{v.\, \boxed{P}^{\mathcal{N}} * Q_2\right\}_{\mathcal{E}}}$$

This rule is quite a mouthful, so we will go over it carefully. First, there is the proposition $\boxed{P}^{\mathcal{N}}$, which states that P (an arbitrary proposition) is maintained as an invariant. The rule says that having this proposition in the context permits us to access the invariant, which involves acquiring ownership of P *before* the verification of e and giving back ownership of P *after* said verification. Crucially, we require that e is *atomic*, meaning that computation is guaranteed to complete in a single step. This is essential for soundness: the rule allows us to temporarily use and even break the invariant, but after a single atomic step (*i.e.,* before any other thread could take a turn), we have to establish it again.

Notice that $\boxed{P}^{\mathcal{N}}$ is *just another kind of proposition*, and it can be used anywhere that normal propositions can be used—including the pre- and postconditions of Hoare triples, and invariants themselves, resulting in nested invariants. The latter property is sometimes

referred to as *impredicativity*. Impredicativity is the reason that the invariant P appears with a "later" modality $\triangleright P$ in the pre- and postcondition; without the later modality, the rule for opening invariants is unsound (see §8.2). However, if one does not store Hoare triples or invariants inside invariants, one can generally ignore the later modality, which we will do throughout this section. We will further discuss the later modality in §5.5.

Finally, we come to the *mask* $\mathcal{E}$ and *namespace* $\mathcal{N}$: they avoid the issue of *reentrancy*. We have to make sure that the same invariant is not accessed twice at the same time, as that would incorrectly duplicate the underlying resource. To this end, each invariant has a *namespace* $\mathcal{N}$ identifying it. Furthermore, each Hoare triple is annotated with a *mask* to keep track of which invariants are still enabled.[7] Accessing an invariant removes its namespace from the mask, ensuring that it cannot be accessed again in a nested fashion.

Invariants are created using the INV-ALLOC rule (Figure 4): Whenever a proposition P has been established, it can be turned into an invariant. This can be viewed as a transfer of the resources backing up P from being locally owned by some thread to being shared by all threads via the invariant. Creating an invariant is a view shift; we will see the reason for this in §7, where we will see how exactly invariants in Iris are working "under the hood". Until then we will ignore the namespaces $\mathcal{N}$ of invariants ($\boxed{P}^{\mathcal{N}}$) and the masks $\mathcal{E}$ of Hoare triples ($\{P\}\ e\ \{v.\ Q\}_{\mathcal{E}}$) and view shifts ($P \Rrightarrow_{\mathcal{E}} Q$).

2.3 Persistent propositions

We have seen that Iris can both express *ownership* of exclusive resources (like $\ell \mapsto v$ or $\boxed{\mathsf{pending}}^{\gamma}$), as well as *knowledge* of properties like $\boxed{P}^{\mathcal{N}}$ and $\boxed{\mathsf{shot}(n)}^{\gamma}$ that, once true, hold true forever. We call the latter class of propositions *persistent*. Further examples of persistent propositions are validity ($\overline{\mathcal{V}}(a)$), equality ($t = u$), Hoare triples ($\{P\}\ e\ \{v.\ Q\}$) and view shifts ($P \Rrightarrow Q$). Persistent propositions can be freely duplicated (PERSISTENT-DUP); the usual restriction of resources being usable only once does not apply to them.

The versatility of ghost ownership is also visible in its relation to persistence: while ghost ownership of some elements (like pending) is ephemeral, ownership *of a core* is persistent (PERSISTENT-GHOST). This manifests the idea mentioned in §2.1 that RAs can express both ownership and knowledge in one unified framework, with knowledge merely referring to ownership of a persistent resource. In this view, the core $|a|$ is a function which extracts the knowledge out of RA element a. In the proof of `check`, persistent ghost ownership will be crucial.

One important role of persistent propositions is related to nested Hoare triples: as expressed by the rule HOARE-CTX, the nested Hoare triple may only use propositions Q from the "outer" context that are persistent. Persistence guarantees that Q will still hold when the Hoare triple is "invoked" (*i.e.,* when the code it specifies is executed), even if that happens multiple times.

A closely related concept is the notion of *duplicable propositions*, *i.e.,* propositions P for which one has $P \Leftrightarrow P * P$. This is a strictly weaker notion, however: not all duplicable

[7] By convention, the absence of a mask means we are using the full mask containing all the namespaces.

$$
\begin{array}{l}
\texttt{let}\, x = \texttt{ref}(\texttt{inl}(0))\, \texttt{in} \\
\left\{x \mapsto \texttt{inl}(0) * \boxed{\texttt{pending}}^{\gamma}\right\} \\
\left\{\boxed{I}^{\mathcal{N}}\right\} \text{where } I \triangleq (x \mapsto \texttt{inl}(0) * \boxed{\texttt{pending}}^{\gamma}) \vee (\exists n.\, x \mapsto \texttt{inr}(n) * \boxed{\texttt{shot}(n)}^{\gamma}) \\
\{\ \texttt{tryset} = \lambda n. \\
\quad \left\{\boxed{I}^{\mathcal{N}}\right\} \\
\quad \{I\}\, \texttt{CAS}(x, \texttt{inl}(0), \texttt{inr}(n))\, \{I\} \\
\quad \left\{\boxed{I}^{\mathcal{N}}\right\} \\
\ \ \texttt{check} = \lambda _. \\
\quad \left\{\boxed{I}^{\mathcal{N}}\right\} \\
\quad \{I\}\, \texttt{let}\, y = !x\, \texttt{in}\ \{I * P\} \text{where } P \triangleq y = \texttt{inl}(0) \vee (\exists n.\, y = \texttt{inr}(n) * \boxed{\texttt{shot}(n)}^{\gamma}) \\
\quad \left\{\boxed{I}^{\mathcal{N}} * P\right\} \\
\qquad \lambda _. \\
\qquad \left\{\boxed{I}^{\mathcal{N}} * P\right\} \\
\qquad \{I * P\} \\
\qquad \texttt{let}\, z = !x\, \texttt{in} \\
\qquad \left\{I * \left(y = \texttt{inl}(0) \vee (\exists n.\, y = z = \texttt{inr}(n))\right)\right\} \\
\qquad \left\{\boxed{I}^{\mathcal{N}} * \left(y = \texttt{inl}(0) \vee (\exists n.\, y = z = \texttt{inr}(n))\right)\right\} \\
\qquad \texttt{match}\, y, z\, \texttt{with} \\
\qquad\ \ \texttt{inl}(_)\,, _ \quad\ \Rightarrow () \\
\qquad \mid \texttt{inr}(n), \texttt{inl}(_) \ \Rightarrow \texttt{assert}(\texttt{false}) \\
\qquad \mid \texttt{inr}(n), \texttt{inr}(m) \Rightarrow \texttt{assert}(n = m) \\
\qquad \texttt{end} \\
\quad \}
\end{array}
$$

Fig. 5. Example proof outline.

propositions are persistent. For example, considering the points-to connective $\ell \overset{q}{\mapsto} v$ with a fractional permission q (Boyland, 2003; Bornat *et al.*, 2005), the proposition $\exists q.\, \ell \overset{q}{\mapsto} v$ is duplicable (which follows from halving the fractional permission q), but it is not persistent.

2.4 Proof of the example

We have now seen enough features of Iris to proceed with the actual verification problem outlined in Figure 2. We show a Hoare outline of the proof in Figure 5. Notice that we let-expanded the load of x in the `match`. This is solely to facilitate writing the proof outline; Iris is perfectly capable of verifying the correctness of the code as given originally.

Proof of `mk_oneshot`. First of all, from the allocation performed by the `ref` construct, we obtain $x \mapsto \texttt{inl}(0)$. Next, we allocate a new ghost location γ with the structure of the oneshot RA defined above, picking the initial state pending. (This is implicitly using HOARE-VS to justify applying a view shift while verifying a Hoare triple.) Finally, we establish and create the following invariant:

$$I \triangleq (x \mapsto \texttt{inl}(0) * \boxed{\texttt{pending}}^{\gamma}) \vee (\exists n.\, x \mapsto \texttt{inr}(n) * \boxed{\texttt{shot}(n)}^{\gamma})$$

Since x is initialized with $\texttt{inl}(0)$, the invariant I initially holds. What remains to be done is establishing our postcondition, which consists of two Hoare triples. Thanks to HOARE-CTX, we can use the freshly allocated invariant for the proofs of these triples. (In the proof outline, HOARE-CTX allows us to keep resources when we "step over a λ", but only when the resources are persistent. This corresponds to the fact that the function can be called several times, so that the resources should not be used up by one call.)

Proof of `tryset`. The function `tryset` accesses location x, so we start by opening the invariant around the CAS. CAS is atomic, so the rule HOARE-INV applies. The invariant provides $x \mapsto _$ no matter which side of the disjunction we obtain, so safety of the memory operation is justified. In case the `CAS` fails, no change is made to x, so reestablishing the invariant is immediate.

The case in which the `CAS` succeeds is more subtle. Here, we know that x originally had value $\texttt{inl}(0)$, so we obtain the invariant in its left disjunct. Thus, after the `CAS`, we have the following:

$$x \mapsto \texttt{inr}(n) * \boxed{\mathsf{pending}}^{\gamma}$$

How can we reestablish the invariant I after this `CAS`? Clearly, we must pick the right disjunct, since $x \mapsto \texttt{inr}(n)$. Hence we have to update the ghost state to match the physical state. To this end, we apply GHOST-UPDATE with the frame-preserving update ONESHOT-SHOOT, which allows us to update the ghost location to $\mathsf{shot}(n)$ if we own $\mathsf{pending}$, which we do. We then have I again and can finish the proof.

Notice that we could *not* complete the proof if `tryset` would ever change x again, since ONESHOT-SHOOT can only ever be used once on a particular ghost location. We have to be in the $\mathsf{pending}$ state if we want to pick the n in $\mathsf{shot}(n)$. This is exactly what we would expect, since `check` indeed relies on x not being modified once it has been set to $\texttt{inr}(n)$.

Proof of `check`. What remains is to prove correctness of `check`. We open our invariant I to justify the safety of $!x$, which is immediate since I always provides $x \mapsto _$, but we will *not* immediately close I again. Instead, we will have to acquire some piece of ghost state that shows that *if* we read an $\texttt{inr}(n)$, then x will not change its value. At this point in the proof, we have the following proposition:

$$x \mapsto y \quad * \quad \left((y = \texttt{inl}(0) * \boxed{\mathsf{pending}}^{\gamma}) \lor (\exists n.\, y = \texttt{inr}(n) * \boxed{\mathsf{shot}(n)}^{\gamma})\right)$$

We use the identity $\mathsf{shot}(n) = \mathsf{shot}(n) \cdot \mathsf{shot}(n)$ with GHOST-OP to show that this logically implies:

$$x \mapsto y \quad * \quad \left((y = \texttt{inl}(0) * \boxed{\mathsf{pending}}^{\gamma}) \lor (\exists n.\, y = \texttt{inr}(n) * \boxed{\mathsf{shot}(n)}^{\gamma} * \boxed{\mathsf{shot}(n)}^{\gamma})\right)$$

which in turn implies:

$$I \quad * \quad \underbrace{(y = \texttt{inl}(0) \lor (\exists n.\, y = \texttt{inr}(n) * \boxed{\mathsf{shot}(n)}^{\gamma}))}_{P}$$

We can thus reestablish the invariant I, but we keep P, the information we gathered about y. The plan is to use this in the proof of the closure that we return.

To do so, we have to show that P is persistent as mandated by HOARE-CTX. We already discussed that $\boxed{\mathsf{shot}(n)}^{\gamma}$ and equalities are persistent, and it is indeed the case that all the

standard connectives preserve persistence. This matches the intuition that, once we observe that x has been set, we can then forever assume it will not change again.

To finish this proof, let us look at the closure returned by `check` in more detail: Again, we will open our invariant to justify the safety of $!x$. Our proposition then is $I * P$. In order to proceed, we now distinguish the possible cases in P and I.

1. **Case 1 (P "on the left"):** We have $I * y = \texttt{inl}(0)$. In this case, the match will always pick the first arm, and there is nothing left to show.
2. **Case 2 (P "on the right", I "on the left"):** We have:

   $$x \mapsto \texttt{inl}(0) * \boxed{\mathsf{pending}}^{\gamma} * (\exists n.\, y = \texttt{inr}(n) * \boxed{\mathsf{shot}(n)}^{\gamma})$$

   Using GHOST-OP, we can obtain $\boxed{\mathsf{pending} \cdot \mathsf{shot}(n)}^{\gamma}$ and hence $\boxed{\lightning}^{\gamma}$, which according to GHOST-VALID is a contradiction.
3. **Case 2 (P "on the right", I "on the right"):** We have:

   $$(\exists m.\, x \mapsto \texttt{inr}(m) * \boxed{\mathsf{shot}(m)}^{\gamma}) * (\exists n.\, y = \texttt{inr}(n) * \boxed{\mathsf{shot}(n)}^{\gamma})$$

   In particular, we obtain $\boxed{\mathsf{shot}(n) \cdot \mathsf{shot}(m)}^{\gamma}$. Using GHOST-VALID, this yields $\overline{\mathcal{V}}(\mathsf{shot}(n) \cdot \mathsf{shot}(m))$, implying $n = m$ by SHOT-AGREE. We are hence left with:

   $$x \mapsto \texttt{inr}(n) * y = \texttt{inr}(n) * \boxed{\mathsf{shot}(n)}^{\gamma}$$

   In particular, we know that we will take the third arm of the match, and the assertion reduces to $\texttt{assert}(n = n)$, *i.e.,* the assertion succeeds.

3 Advanced ghost state constructions

In the previous section we have seen that user-defined ghost state plays an important role in Iris. In this section, we will have a more thorough look at ghost state. First of all, in §3.1, we show that many frequently needed RAs can be constructed by composing smaller, reusable pieces. In §3.2 we then show that the ownership connective $\boxed{a}^{\gamma}$ can in fact be defined in terms of an even more primitive notion of "global" ghost ownership. In §3.3 we introduce the more advanced notion of *higher-order ghost state* and show that in its naive form it is inconsistent.

3.1 RA constructions

One of the key features of Iris is that it leaves the structure of ghost state entirely up to the user of the logic. If there is the need for some special-purpose RA, the user has the freedom to directly use it. However, it turns out that many frequently needed RAs can be constructed by composing smaller, reusable pieces—so while we have the entire space of RAs available when needed, we do not have to construct custom RAs for every new proof.

For example, looking at the oneshot RA from §2.1, it really does three things:

1. It separates the allocation of an element of the RA from the decision about what value to store there (ONESHOT-SHOOT).
2. While the oneshot location is uninitialized, ownership is exclusive, *i.e.,* at most one thread can own the location.

3. Once the value has been decided on, it makes sure everybody agrees on that value.

We can thus decompose the oneshot RA into the *sum*, *exclusive* and *agreement* RAs as described below. (In the definitions of all the RAs, the omitted cases of the composition and core are all $\lightning$.)

Sum. The *sum RA* $M_1 +_{\lightning} M_2$ for any RAs M_1 and M_2 is:

$$M_1 +_{\lightning} M_2 \triangleq \mathsf{inl}(a_1 : M_1) \mid \mathsf{inr}(a_2 : M_2) \mid \lightning$$
$$\overline{\mathcal{V}}(a) \triangleq (\exists a_1 \in M_1.\, a = \mathsf{inl}(a_1) \land \overline{\mathcal{V}}_1(a_1)) \lor (\exists a_2 \in M_2.\, a = \mathsf{inr}(a_2) \land \overline{\mathcal{V}}_2(a_2))$$
$$\mathsf{inl}(a_1) \cdot \mathsf{inl}(a_2) \triangleq \mathsf{inl}(a_1 \cdot a_2) \qquad \mathsf{inr}(a_1) \cdot \mathsf{inr}(a_2) \triangleq \mathsf{inr}(a_1 \cdot a_2)$$
$$|\mathsf{inl}(a_1)| \triangleq \begin{cases} \bot & \text{if } |a_1| = \bot \\ \mathsf{inl}(|a_1|) & \text{otherwise} \end{cases} \qquad |\mathsf{inr}(a_2)| \triangleq \begin{cases} \bot & \text{if } |a_2| = \bot \\ \mathsf{inr}(|a_2|) & \text{otherwise} \end{cases}$$

Exclusive. Given a set X, the task of the *exclusive RA* $\textsc{Ex}(X)$ is to make sure that one party *exclusively* owns a value $x \in X$. We define Ex as follows:

$$\textsc{Ex}(X) \triangleq \mathsf{ex}(x : X) \mid \lightning$$
$$\overline{\mathcal{V}}(a) \triangleq a \neq \lightning$$
$$|\mathsf{ex}(x)| \triangleq \bot$$

Composition is always $\lightning$ to ensure that ownership is exclusive—just like the composition of pending with anything else was $\lightning$ (PENDING-EXCL).

Agreement. Given a set X, the task of the *agreement RA* $\textsc{Ag}_0(X)$ is to make sure multiple parties can *agree* upon which value $x \in X$ has been picked. (We call this $\textsc{Ag}_0$ because we will refine this definition in §4.3 to obtain the final version, Ag.) We define $\textsc{Ag}_0$ as follows:

$$\textsc{Ag}_0(X) \triangleq \mathsf{ag}_0(x : X) \mid \lightning$$
$$\overline{\mathcal{V}}(a) \triangleq \exists x \in X.\, a = \mathsf{ag}_0(x)$$
$$\mathsf{ag}_0(x) \cdot \mathsf{ag}_0(y) \triangleq \begin{cases} \mathsf{ag}_0(x) & \text{if } x = y \\ \lightning & \text{otherwise} \end{cases}$$
$$|\mathsf{ag}_0(x)| \triangleq \mathsf{ag}_0(x)$$

In particular, agreement satisfies the following property corresponding to SHOT-AGREE:

$$\overline{\mathcal{V}}(\mathsf{ag}_0(x) \cdot \mathsf{ag}_0(y)) \Rightarrow x = y \qquad \text{(AG0-AGREE)}$$

Oneshot. We can now define the general idea of the oneshot RA as $\textsc{OneShot}(X) \triangleq \textsc{Ex}(1) +_{\lightning} \textsc{Ag}_0(X)$, and recover the RA for the example as $\textsc{OneShot}(\mathbb{Z})$.

Frame-preserving updates. Another advantage of decomposing RAs into separate pieces is that for these pieces we can prove generic frame-preserving updates. We have the fol-

lowing generic frame-preserving updates for sum and exclusive:

$$\textsc{inl-update}\ \frac{a \rightsquigarrow B}{\mathsf{inl}(a) \rightsquigarrow \{\mathsf{inl}(b) \mid b \in B\}} \qquad \textsc{inr-update}\ \frac{a \rightsquigarrow B}{\mathsf{inr}(a) \rightsquigarrow \{\mathsf{inr}(b) \mid b \in B\}} \qquad \textsc{ex-update}\ \mathsf{ex}(x) \rightsquigarrow \mathsf{ex}(y)$$

The agreement RA permits no non-trivial frame-preserving updates.

Considering the oneshot RA, the above rules are not yet sufficient. In order to prove a variant of the frame-preserving update ONESHOT-SHOOT, as needed for the running example, we need to change the left summand into the right:

$$\mathsf{inl}(\mathsf{ex}()) \rightsquigarrow \mathsf{inr}(\mathsf{ag}_0(x))$$

This frame-preserving update can be proven because $\mathsf{inl}(\mathsf{ex}())$ has *no* frame, *i.e.,* there is no c with $\overline{\mathcal{V}}(\mathsf{inl}(\mathsf{ex}()) \cdot c)$. The absence of a frame allows us to pick any valid RA element on the right-hand side. It turns out that we can easily generalize this idea to any RA: we say that an RA element a is *exclusive* (or has *no frame*) if:

$$\mathrm{exclusive}(a) \triangleq \forall c.\, \neg\overline{\mathcal{V}}(a \cdot c)$$

For exclusive elements we have the following generic frame-preserving update:

$$\textsc{exclusive-update}\ \frac{\mathrm{exclusive}(a) \qquad \overline{\mathcal{V}}(b)}{a \rightsquigarrow b}$$

Notice that the above frame-preserving update is only useful for an RA with a partial core: $\mathrm{exclusive}(a)$ is always false for any valid RA element a with $|a| \neq \bot$. Obtaining frame-preserving updates for switching the "side" of a sum is one of our key motivations for making the core a partial function instead of a total function.

3.2 Derived forms and the global ghost state

In Iris, there is a strong emphasis on only providing a *minimal* core logic, and deriving as much as possible *within* the logic rather than baking it in as a primitive. For example, both Hoare triples and propositions of the form $l \mapsto v$ are actually derived forms. As we will see in §4 and §5, this has the advantage that the model can be kept simpler, since it only has to justify soundness of a minimal core logic.

In this section we discuss the encoding of the proposition $\boxed{a}^{\gamma}$ for ghost ownership, which is not a built-in notion either. As we have seen, this proposition allows one to have *multiple* ghost locations γ, all of which can range over *different* RAs. As a primitive, Iris provides just a *single* global ghost location whose structure is described by a *single* global RA picked by the user. However, by picking an appropriate RA, one can *define* the proposition $\boxed{a}^{\gamma}$ in Iris and *derive* its rules as given in Figure 4 within the logic.

Iris's primitive construct for ghost ownership is $\mathsf{Own}(a)$, whose rules are given in Figure 6. It is worth noting that the global RA must be *unital*, which means it should have a unit element ε (Figure 3). The reason for this is twofold. First of all, it allows us to have the rule OWN-UNIT, which is used to prove GHOST-ALLOC. Second of all, unital RAs enjoy the properties that the extension order $\preccurlyeq$ is reflexive and that the $|-|$ function is total, which

$$\textsc{own-op}\quad \mathsf{Own}(a \cdot b) \Leftrightarrow \mathsf{Own}(a) * \mathsf{Own}(b) \qquad \textsc{own-unit}\quad \mathsf{True} \Rightarrow \mathsf{Own}(\varepsilon) \qquad \textsc{own-core}\quad \mathsf{Own}(|a|) \Leftrightarrow \Box\mathsf{Own}(|a|)$$

$$\textsc{own-valid}\quad \mathsf{Own}(a) \Rightarrow \overline{\mathcal{V}}(a) \qquad \textsc{own-update}\quad \frac{a \rightsquigarrow B}{\mathsf{Own}(a) \Rrightarrow \exists b \in B.\, \mathsf{Own}(b)}$$

Fig. 6. Primitive rules for ghost state.

simplify the model construction in §4 and §5. Note that one can always turn an RA into a unital RA by extending it with a unit element, preserving all frame-preserving updates.

In order to define the $\boxed{a}^{\gamma}$ connective, we need to instantiate the single global ghost state RA with a *heap* of ghost cells. To this end, we assume that we are given a family of RAs $(M_i)_{i\in\mathcal{I}}$ for some index set $\mathcal{I}$, and then we define the RA M of the global ghost state to be the indexed (dependent) product over "heaps of M_i" as follows:

$$M \triangleq \prod_{i\in\mathcal{I}} \mathbb{N} \xrightarrow{\text{fin}} M_i$$

In this construction, we use the natural pointwise lifting of the RA operations from each M_i through the finite maps and products all the way to M, so that M is an RA itself. In fact, it is a uRA, with the unit being the product of all empty maps:

$$\varepsilon \triangleq \lambda j.\, \emptyset$$

It is straightforward to show that frame-preserving updates can be performed pointwise in products, and the following laws hold for frame-preserving updates in finite maps:

$$\textsc{fmap-update}\quad \frac{a \rightsquigarrow B}{f[i \leftarrow a] \rightsquigarrow \{f[i \leftarrow b] \mid b \in B\}} \qquad \textsc{fmap-alloc}\quad \frac{a \in \overline{\mathcal{V}}}{f \rightsquigarrow \{f[i \leftarrow a] \mid i \notin \mathrm{dom}(f)\}}$$

Notice that FMAP-ALLOC is a *non-deterministic* frame-preserving update: we cannot pick the index i of the new element in advance because it depends on the frame. Instead we are guaranteed to obtain a new singleton map with *some* fresh index i being mapped to a. The rule FMAP-UPDATE expresses pointwise lifting of updates to the finite map.

Using M as the uRA to instantiate Iris with allows us to (a) use all the M_i in our proofs, and (b) treat ghost state as a heap, where we can allocate new instances of any of the M_i at any time. We define the connective for ghost ownership of a single location as:

$$\boxed{a : M_i}^{\gamma} \triangleq \mathsf{Own}\left(\lambda j.\, \begin{cases} [\gamma \mapsto a] & \text{if } i = j \\ \emptyset & \text{otherwise} \end{cases}\right)$$

In other words, $\boxed{a : M_i}^{\gamma}$ asserts ownership of the singleton heap $[\gamma \mapsto a]$ at position i in the product. We typically leave the concrete M_i implicit and write just $\boxed{a}^{\gamma}$. The rules for $\boxed{-}^{-}$ given in Figure 4 can now be derived from those for $\mathsf{Own}(-)$ shown in Figure 6.

Obtaining modular proofs. Even with multiple RAs at our disposal, it may still seem like we have a modularity problem: every proof is done in an instantiation of Iris with some

particular family of RAs. As a result, if two proofs make different choices about the RAs, they are carried out in entirely different logics and hence cannot be composed.

To solve this problem, we *generalize* our proofs over the family of RAs that Iris is instantiated with. All proofs are carried out in Iris instantiated with some unknown $(M_i)_{i\in\mathcal{I}}$. If the proof needs a particular RA, it further assumes that there exists some j such that M_j is the desired RA. Composing two proofs is thus straightforward; the resulting proof works in any family of RAs that contains all the particular RAs needed by either proof. Finally, if we want to obtain a "closed form" of some particular proof in a concrete instance of Iris, we simply construct a family of RAs that contains only those that the proof needs.

Notice that the generalization over resource algebras happens at the *meta-level* here, so the set of available resource algebras must be fixed before the proof begins. If, for example, the type of the ghost state should depend on some program value determined only at runtime, that would involve a form of dependent type. We have not yet explored to what extent our approach is compatible with this situation, as it does not seem to arise in practice.

3.3 Naive higher-order ghost state paradox

As we have seen, the setup of Iris is that the user provides a resource algebra M by which the logic is parameterized. This gives a lot of flexibility, as the user is free to decide what kind of ghost state to use. We now discuss whether we could stretch this flexibility even further, by allowing the construction of the resource algebra M to depend on the type of Iris propositions *iProp*. In Jung *et al.* (2016), we dubbed this phenomenon *higher-order ghost state* and showed that it has useful applications in program verification. In Krebbers *et al.* (2017a), we then showed that higher-order ghost state has applications beyond program verification: it can be used to encode invariants in terms of plain ghost state.

The way we model higher-order ghost state puts certain restrictions on the way *iProp* can be used in the construction of the user-supplied resource algebra. In Jung *et al.* (2016), these restrictions appeared as a semantic artifact of modeling the logic. However, in this section we show that restrictions of some kind are in fact necessary: allowing higher-order ghost state in a naive or unrestricted manner leads to a paradox (*i.e.,* an unsound logic). In order to demonstrate this paradox, we use the simplest instance of higher-order ghost state, *named propositions* (also called "stored" or "saved" propositions) (Dodds *et al.*, 2016).

In order to explain named propositions, let us have another look at the agreement resource algebra $\mathrm{AG}_0(X)$. As part of the decomposition of the oneshot RA into small, reusable pieces, we also generalized the set of values that the proof can pick from (*i.e.,* the parameter of AG_0) from $\mathbb{Z}$ to any given set X. Instead of choosing X to be a "simple" set like $\mathbb{Z}$ or $\wp(\mathbb{N})$, we could try choosing it to be *iProp*, the type of Iris propositions. The RA $\mathrm{AG}_0(\mathit{iProp})$ would then allow us to associate a ghost location (or "name") γ with a proposition P. However, this ability to "name" propositions leads to a paradox.

Theorem 1 (Higher-order ghost state paradox)**.** *Assume we can instantiate the construction from §3.2 with* $\mathrm{AG}_0(\mathit{iProp})$ *being in the family of RAs. Then we have:*

$$\mathit{True} \Rrightarrow_{\mathcal{E}} \mathit{False}$$

Before proving this paradox, let us take a look at the proof rules of Iris that are crucial to prove the paradox:

$$\mathsf{True} \Rrightarrow \exists \gamma.\, \boxed{\mathsf{ag}_0(A(\gamma))}^{\gamma} \qquad \text{(SPROP0-ALLOC)}$$

$$\boxed{\mathsf{ag}_0(P_1)}^{\gamma} * \boxed{\mathsf{ag}_0(P_2)}^{\gamma} \Rightarrow P_1 = P_2 \qquad \text{(SPROP0-AGREE)}$$

The rule SPROP0-AGREE, which states that named propositions with the same name are the same, follows easily from AG0-AGREE. The rule SPROP0-ALLOC, which allows one to allocate a name for a proposition, looks a lot like an instance of GHOST-ALLOC, except that the initial state of the new ghost location may depend on the fresh γ (here, A is a function mapping ghost names to *iProp*, which lets us express said dependency). In fact, the rule SPROP0-ALLOC follows from the following generalized rule for ghost state allocation:

$$\frac{\forall \gamma.\, \overline{\mathcal{V}}(g(\gamma))}{\mathsf{True} \Rrightarrow_{\mathcal{E}} \exists \gamma.\, \boxed{g(\gamma)}^{\gamma}} \qquad \text{(GHOST-ALLOC-DEP)}$$

This rule allows the initial state of the freshly allocated ghost location (described by the function g) to depend on the location γ that has been picked. The rule GHOST-ALLOC-DEP can be proved from the basic rules for the $\mathsf{Own}(-)$ connective as shown in Figure 6 and the following frame-preserving update on finite maps:

$$\frac{\forall i.\, \overline{\mathcal{V}}(g(i))}{f \rightsquigarrow \{f[i \leftarrow g(i)] \mid i \notin \mathrm{dom}(f)\}} \qquad \text{(FMAP-ALLOC-DEP)}$$

This completes the proof of SPROP0-ALLOC.

We now turn towards the proof of Theorem 1. First, we define

$$A(\gamma) \triangleq \exists P.\, \square(P \Rightarrow \mathsf{False}) * \boxed{\mathsf{ag}_0(P)}^{\gamma}$$

$$Q(\gamma) \triangleq \boxed{\mathsf{ag}_0(A(\gamma))}^{\gamma}$$

Intuitively, $A(\gamma)$ asserts that ghost location γ names some proposition P that does not hold. (Since our ghost locations are described by the RA $\mathrm{AG}_0(\mathit{iProp})$, we know that everybody agrees on which proposition is named by a ghost location γ, so it makes sense to talk about "the proposition with name γ".)

The proposition $Q(\gamma)$ says that the proposition with name γ is $A(\gamma)$. After unfolding the definition of A, we can see that $Q(\gamma)$ means that the proposition with name γ says that the proposition with name γ does not hold—*i.e.,* the proposition with name γ states its own opposite. This is an instance of the classic "liar's paradox", so it should not be surprising that it leads to a contradiction. Concretely, we show the following lemma, from which Theorem 1 is a trivial consequence.

Lemma 1. *We have the following properties:*

1. $Q(\gamma) \Rightarrow \square(A(\gamma) \Rightarrow \mathit{False})$,
2. $Q(\gamma) \Rightarrow A(\gamma)$, *and,*
3. $\mathit{True} \Rrightarrow_{\mathcal{E}} \exists \gamma.\, Q(\gamma)$.

Proof. Notice that all propositions we are using for this paradox, except the existentially quantified P in the definition of $A(\gamma)$, are persistent, so we can mostly ignore the substructural aspects of Iris in this proof.

For 1, we can assume $Q(\gamma)$ and $A(\gamma)$ and we have to show False. After unfolding both of them, we obtain some P such that:

$$\boxed{\mathsf{ag}_0(A(\gamma))}^{\gamma} * \Box(P \Rightarrow \mathsf{False}) * \boxed{\mathsf{ag}_0(P)}^{\gamma}$$

From SPROP0-AGREE, we can now obtain that $A(\gamma) = P$. Since we have a proof of $A(\gamma)$, and we have $\Box(P \Rightarrow \mathsf{False})$, this yields the desired contradiction.

For 2, we can assume $Q(\gamma)$. Our goal is $A(\gamma)$, so we start by picking $P \triangleq A(\gamma)$. We now have to show $\Box(A(\gamma) \Rightarrow \mathsf{False})$, which easily follows from (a) and our $Q(\gamma)$. We further have to show $\boxed{\mathsf{ag}_0(A(\gamma))}^{\gamma}$, which is exactly $Q(\gamma)$. Hence we are done.

Finally, for 3, we use SPROP0-ALLOC. □

So, what went wrong here? Where does this contradiction originate from? The problem is that higher-order ghost state, and named propositions in particular, allow us to express concepts like "the proposition with name γ". This allows us to define a proposition as being its own negation. Usually, such nonsense is prevented because the definition of a proposition cannot refer back to this proposition in a cyclic way; the fact that higher-order ghost state lets us give names to propositions lets us circumvent this restriction and close the cycle. In some sense, this is very similar to Landin's knot, where a recursive function is created by indirecting recursive references of the function to itself through a location on the (higher-order) heap.

The paradox shows that one has to be very careful if one wishes to support higher-order ghost state. In particular, one cannot allow resource algebras like $\mathrm{AG}_0(\mathit{iProp})$ to be used to instantiate the Iris logic. In the subsequent sections of this paper we show that higher-order ghost state is sound if the recursive occurrences of *iProp* are suitably *guarded*. To model named propositions, one should then use $\mathrm{AG}(\blacktriangleright \mathit{iProp})$. Here, AG is a refined definition of the agreement RA, and the "later" $\blacktriangleright$ plays the role of a guard. As a result of the $\blacktriangleright$ guard (whose constructor is next), we only get a weaker version of the rule SPROP0-AGREE:

$$\boxed{\mathsf{ag}(\mathsf{next}(P_1))}^{\gamma} * \boxed{\mathsf{ag}(\mathsf{next}(P_2))}^{\gamma} \Rightarrow \triangleright(P_1 = P_2)$$

This rule gives us the equality one step of execution "later", which prohibits the paradox in this section. In the next sections we show how to extend the logic and its model with the operators $\blacktriangleright$ and $\triangleright$.

4 A model of Iris

In the previous sections, we gave a high-level tour of Iris. In the remainder of the paper, we will make the definition of Iris more precise and prove that it is sound.

As in much prior work on separation logic, we prove soundness of Iris by giving a *semantic model*. Fundamentally, such a proof proceeds in three steps:

1. Define a *semantic domain* of propositions with an appropriate notion of entailment.
2. Define an *interpretation function* that maps all statements in the logic to elements of the semantic domain.
3. For every proof rule of the logic, show that the semantic interpretation of the premise entails the semantic interpretation of the conclusion.

In this section, we tackle (1): defining the semantic domain of Iris propositions. This may seem somewhat difficult since we have not yet nailed down exactly what Iris's propositions are. However, in §2, we have already given an intuition for the kind of statements we want to make in Iris. This is sufficient to discuss the construction of the semantic domain of Iris propositions. We will then proceed bottom-up, first in §5 building the Iris base logic on top of this semantic domain, then in §6 and §7 building the higher-level program logic connectives that we saw in §2 on top of the base logic.

As demonstrated in §2, one of the main features of Iris is resource ownership, which, for a separation logic, should not be very surprising. The usual way (O'Hearn *et al.*, 2001) to model the propositions of a separation logic is as *predicates* over resources. Whereas traditionally "resource" meant a fragment of the heap, in our case, "resource" is something the user can define by an RA of their choice. Furthermore, since Iris is an affine separation logic (see also §9.5), these predicates have to be monotone: If a predicate holds of a resource, it should hold of any larger resource.[8]

This leads us to the following attempt at a semantic domain of Iris propositions:

$$iProp \triangleq Res \xrightarrow{\text{mon}} Prop$$

Here, *Res* is the global resource algebra, chosen by the user, and *Prop* is the domain of propositions of the ambient meta-logic.

However, recall that we mentioned in §3.3 that we will need *higher-order ghost state* to encode advanced features of Iris like invariants. That means we want the resources *Res* to depend on *iProp*, while at the same time *iProp* depends on *Res*!

In order to formally capture this dependency, we replace the fixed choice of *Res* by a function F that can depend on *iProp*. The user can then pick F (rather than picking *Res*), and *Res* is defined as $F(iProp)$. Thus, the definition changes as follows:

$$iProp = Res \xrightarrow{\text{mon}} Prop \qquad \text{where} \qquad Res \triangleq F(iProp) \tag{PRE-IRIS}$$

Unfortunately, this is no longer just a definition—the existence of a solution *iProp* to this equation is not at all obvious. In fact, for certain choices of F, for example by taking $F(X) \triangleq \text{AG}_0(X)$, we end up with $iProp = \text{AG}_0(iProp) \xrightarrow{\text{mon}} Prop$, which can be shown to have no solutions using a cardinality argument. This should not be surprising; after all, in §3.3 we showed that having $\text{AG}_0(iProp)$ as our type of resources leads to a contradiction in the logic, which implies that no model with $\text{AG}_0(iProp)$ as the type of resources can exist!

In order to circumvent this problem, we use *step-indexing* (Appel & McAllester, 2001). Roughly, the idea of step-indexing is to make elements of *iProp* a *sequence* of propositions: the n-th proposition in such a sequence holds as far as only the n following steps of computation are concerned. This *step-index* can be used for stratifying the recursion in PRE-IRIS. In the following, we give a conceptual overview of the model and the use of step-indexing in §4.1. Subsequently, in §4.2-§4.4, we introduce the actual infrastructure to treat step-indexing in a modular way. Finally, in §4.5-§4.7, we use this infrastructure to obtain a semantic domain of Iris propositions.

[8] RAs are ordered by the extension order $\preccurlyeq$ (defined in Figure 3), and *Prop* is ordered by entailment.

4.1 Informal and conceptual overview of the model construction

The precise definitions in the following sections are somewhat technical and may somewhat obscure the conceptual simplicity at the heart of the model. Hence, we first give an informal overview of how the model can be understood and how the technical definitions used in the following sections are really systematic and natural generalizations of concepts we have already seen.

Since (Birkedal *et al.*, 2011) it is known that step-indexing can be treated in an abstract, categorical way. Roughly speaking, this means that we can apply the following translation of terminology to go from the usual set-theoretical setting to a step-indexed setting:

Set-theoretic setting	Step-indexed setting
Sets	OFEs (Ordered families of equivalences)
Prop (Propositions)	*SProp* (Downwards-closed sets of step-indices)
Functions	Non-expansive functions

The notions on the right-hand side of the table will be explained in §4.2. For now, it is important to know that the usual operations on *Prop* (*e.g.,* equality, conjunction, implication and quantification) and constructions on sets (*e.g.,* products and sums) are also available on *SProp* and OFEs, respectively. These operations and constructions furthermore enjoy many of the same laws we know from set theory, which means that most of the time, one can entirely ignore in which of these two settings we are working.

Notice that not just the notions of propositions and sets change, but also the function space: In the step-indexed setting, we are restricted to the *non-expansive* functions. These are functions that are "compatible" with the structure that OFEs impose on their domains and codomains. By restricting the function space, we get that the proposition $\forall x, y.\, x = y \Rightarrow f(x) = f(y)$ in *SProp* (*i.e.,* after translation to the step-indexed setting) always holds—in fact, this is the very definition of non-expansiveness of f.

If set theory and the step-indexed setting behave much the same, what is the benefit of working in the step-indexed setting? It turns out that unlike in the set-theoretic setting, in the step-indexed setting one *can* obtain a solution (up to isomorphism) to the equation PRE-IRIS that we have seen before:

$$iProp \cong Res \xrightarrow{\text{mon}} SProp \qquad \text{where} \qquad Res \triangleq F(iProp)$$

The catch to solving this equation in the step-indexed setting is that F has to be "guarded" in some sense (by a "later" $\blacktriangleright$ as we will see in this section).

While the Iris model presented in this section may seem arbitrary and the reader may have the impression that everything just "luckily" works out in the end, this is not the case. The definitions arise by translating a standard set-theoretic model of the logic of bunched implications (O'Hearn & Pym, 1999) to the step-indexing world. This translation works out in the expected way due to the close correspondence between the set-theoretic and the step-indexed setting. However, showing that one can work in the step-indexed setting without even noticing that one is doing so requires a significant body of category theory. In this paper, we will avoid diving into category theory, and thus present the model in elementary terms instead of categorically. Essentially, we "unfold" the step-indexed definitions so that we can stay in set theory as our ambient framework.

An *ordered family of equivalences* (OFE) is a tuple $(T, (\overset{n}{=} \subseteq T \times T)_{n\in\mathbb{N}})$ satisfying:

$$\forall n.\ (\overset{n}{=}) \text{ is an equivalence relation} \qquad \text{(OFE-EQUIV)}$$

$$\forall n, m.\ n \geq m \Rightarrow (\overset{n}{=}) \subseteq (\overset{m}{=}) \qquad \text{(OFE-MONO)}$$

$$\forall x, y.\ x = y \Leftrightarrow (\forall n.\ x \overset{n}{=} y) \qquad \text{(OFE-LIMIT)}$$

A *complete OFE* (COFE) is an OFE T where:

$$\forall (c_n)_{n\in\mathbb{N}}.\ (\forall n, m.\ n \leq m \Rightarrow c_m \overset{n}{=} c_n) \Rightarrow \exists c_\infty.\ \forall n.\ c_n \overset{n}{=} c_\infty \qquad \text{(OFE-COMPL)}$$

A function $f : T \to U$ is *non-expansive* if:

$$\forall n, x, y.\ x \overset{n}{=} y \Rightarrow f(x) \overset{n}{=} f(y) \qquad \text{(OFE-NONEXP)}$$

A function $f : T \to U$ is *contractive* if:

$$\forall n, x, y.\ (\forall m < n.\ x \overset{m}{=} y) \Rightarrow f(x) \overset{n}{=} f(y) \qquad \text{(OFE-CONTR)}$$

Fig. 7. (C)OFEs and non-expansive/contractive functions.

Presenting the model in elementary terms not only makes the construction more accessible, but it also eases the formalization in Coq. The formalization of the Iris model in Coq is thus very close to the presentation in this section.

4.2 Ordered families of equivalences

Ordered families of equivalences (OFEs), proposed by Di Gianantonio & Miculan (2002) and defined in Figure 7, are sets with some basic algebraic structure to equip them with a form of step-indexing. The key intuition behind OFEs is that elements x and y are n-equivalent, written $x \overset{n}{=} y$, if they are *equivalent for n steps of computation*, *i.e.,* if they cannot be distinguished by a program running for no more than n steps. It follows that, as n increases, $\overset{n}{=}$ becomes more and more refined (OFE-MONO). In the limit, it agrees with plain equality (OFE-LIMIT).

A function $f : T \overset{\text{ne}}{\to} U$ between two OFEs is ***n**on-**e**xpansive* if it preserves the structure defined by $(\overset{n}{=})$ (OFE-NONEXP). In other words, applying such a function to some data will not suddenly introduce differences between seemingly equal data. Elements that cannot be distinguished by programs within n steps remain indistinguishable after applying f.

In the step-indexed world, we will always work with equality up to some n. For these equalities to be of any use, it is hence crucial that all functions respect the equivalence relation $\overset{n}{=}$. This is why we require all functions to be non-expansive.

If f does not just preserve n-equality, but actually makes things "more equal", then we say that f is *contractive* (OFE-CONTR). Notice that the definition given in Figure 7 is equivalent to demanding $f(x) \overset{0}{=} f(y)$ and $x \overset{n}{=} y \Rightarrow f(x) \overset{n+1}{=} f(y)$.

The OFEs that are essential to the Iris model are defined in Figure 8. The *discrete OFE* ΔX turns any set X into an OFE by assigning the degenerate step-indexed equivalence. The *later* OFE $\blacktriangleright T$ moves the step-indexed equivalence of an OFE T up by one, thus making everything "one level more equal" than in T.

The OFE of *step-indexed* propositions *SProp* represents propositions ("truth values") that are *true for some number of steps*, or forever. (*SProp* corresponds to the set of ordinals up to and including ω.) Crucially, elements of *SProp* are *downwards-closed*, which means

$$
\begin{array}{rrl}
\textbf{Step-indexed propositions:} & \mathit{SProp} \triangleq & \{X \in \wp(\mathbb{N}) \mid \forall n \geq m.\, n \in X \Rightarrow m \in X\} \\
& X \overset{n}{=} Y \triangleq & \forall m \leq n.\, m \in X \Leftrightarrow m \in Y \\
& X \overset{n}{\subseteq} Y \triangleq & \forall m \leq n.\, m \in X \Rightarrow m \in Y \\
\textbf{Discrete:} & \Delta X \triangleq & X \\
& x \overset{n}{=} y \triangleq & x = y \\
\textbf{Later:} & \blacktriangleright T \triangleq & \{\mathsf{next}(x) \mid x \in T\} \\
& \mathsf{next}(x) \overset{n}{=} \mathsf{next}(y) \triangleq & n = 0 \vee x \overset{n-1}{=} y
\end{array}
$$

Fig. 8. The OFEs used in constructing the Iris model.

that if a statement is valid for n steps, it is also valid for $m \leq n$ steps. Instead of just working with absolute ideas of things being either true or not, *SProp* provides a more approximate approach. Fully valid statements are represented by $\mathbb{N}$, false statements are represented by $\emptyset$; but there are also infinitely many "shades of truth" between these two extremes. Two step-indexed propositions are equal at step-index n if they agree for all step-indices up until and including n. Similarly, we define step-indexed inclusion $(\overset{n}{\subseteq})$ as inclusion ignoring elements larger than n.

An important class of step-indexed propositions are equalities between elements of an OFE. Indeed, the set $\left\{n \;\middle|\; x \overset{n}{=} y\right\}$ of step-indices at which x and y are equal is an *SProp*: From OFE-MONO, it follows that the closure requirement of *SProp* is satisfied. Now, if f is a non-expansive function, then $\left\{n \;\middle|\; x \overset{n}{=} y\right\} \subseteq \left\{n \;\middle|\; f(x) \overset{n}{=} f(y)\right\}$, which means that non-expansiveness of f allows us to perform rewriting of a step-indexed equality under f.

Chains and complete OFEs. The step-indexed equality of an OFE gives rise to the notion of a *chain*, which is a sequence $(c_n)_{n\in\mathbb{N}}$ of elements that become "more and more equal":

$$\forall n, m.\, n \leq m \Rightarrow c_m \overset{n}{=} c_n$$

A chain has a *limit* if there exists an element c_∞ that is "more and more equal" to c_n as n increases:

$$\forall n.\, c_n \overset{n}{=} c_\infty$$

An OFE T is said to be *complete* (it is a *COFE*) if every chain in T has a limit (OFE-COMPL).

Chains with limits are interesting because they can be used to define an object as the limit of a sequence of approximations, rather than defining the object directly. For example, we will see in §5.5 how this can be used to take the fixed-point of a function. All of the OFEs discussed so far are complete.

4.3 Higher-order agreement

With the framework of the model—in particular the notion of an OFE—in hand, let us now look again at the agreement RA AG_0 defined in §3.1. In §3.3, we explained that we would like to obtain an agreement construction on *iProp* in order to encode higher-order ghost state, and we explained in the beginning of the present section that doing this naively leads to a circularity. Let us however ignore that problem for now, and instead have a closer look

at $\mathrm{AG}_0(T)$ for any OFE T. Our goal is to make this "work" as an OFE, *i.e.,* $\mathrm{AG}_0(T)$ should itself be a well-behaved OFE.

Equipping $\mathrm{AG}_0(T)$ with a step-indexed equality is straightforward (we just lift the equality from T). However, there is a problem: the composition operator $(\cdot)$ is *not* non-expansive. That is, given P and Q that are equal up to n steps, but not equal for all steps, *i.e.,* $P \overset{n}{=} Q$ for some n while $P \neq Q$, we have:

$$\mathsf{ag}(P) \cdot \mathsf{ag}(Q) = \lightning \quad \overset{n}{\neq} \quad \mathsf{ag}(P) = \mathsf{ag}(P) \cdot \mathsf{ag}(P)$$

For $(\cdot)$ to be non-expansive, we should have $\mathsf{ag}(P) \cdot \mathsf{ag}(P) \overset{n}{=} \mathsf{ag}(P) \cdot \mathsf{ag}(Q)$, which is clearly not the case. This is not entirely surprising: Agreement composition was defined in terms of plain (absolute) equality rather than step-indexed equality.

In order to maintain the correspondence laid out in §4.1, we want composition to be non-expansive. This means we have to find another construction. Given any OFE T, we are looking for an OFE and RA $\mathrm{AG}(T)$ with a non-expansive composition. Just like the previous construction, AG has to provide a non-expansive function $\mathsf{ag} \in T \xrightarrow{\mathrm{ne}} \mathrm{AG}(T)$ and they should enjoy the following properties:

$$\forall a \in \mathrm{AG}(T).\ |a| = a \tag{AG-CORE}$$

$$\forall x.\ \overline{\mathcal{V}}(\mathsf{ag}(x)) \tag{AG-VALID}$$

$$\forall x, y.\ \overline{\mathcal{V}}(\mathsf{ag}(x) \cdot \mathsf{ag}(y)) \Rightarrow x = y \tag{AG-AGREE}$$

These properties imply $a \cdot a = a$ and injectivity of ag.

Any construction satisfying these laws is sufficient to perform the proof in §2.4. However, that is not enough. Our goal is to have a construction that interacts well with step-indexing. As already mentioned, in a step-indexed world, we are dropping absolute terms like the elements being *(exactly) equal* or statements being *(completely) valid*. Instead we use *step-indexed* versions of those terms. So what we are looking for is a step-indexed version of AG-CORE, AG-VALID and AG-AGREE. To this end, we first turn validity into a step-indexed predicate $\mathcal{V}$ of type $\mathrm{AG}(T) \to \mathit{SProp}$. (This also explains why we used $\overline{\mathcal{V}}$ up to now: We kept the identifier $\mathcal{V}$ free for the more general step-indexed validity.) Now we can formulate the following properties:

$$\forall a \in \mathrm{AG}(T), n.\ |a| \overset{n}{=} a \tag{AG-CORE-STEP}$$

$$\forall x, n.\ n \in \mathcal{V}(\mathsf{ag}(x)) \tag{AG-VALID-STEP}$$

$$\forall x, y, n.\ n \in \mathcal{V}(\mathsf{ag}(x) \cdot \mathsf{ag}(y)) \Rightarrow x \overset{n}{=} y \tag{AG-AGREE-STEP}$$

The new $\mathcal{V}$ and these properties arise by applying the translation outlined in §4.1 to AG-CORE, AG-VALID and AG-AGREE.

First, we mention some key observations:

- The rule AG-CORE-STEP is equivalent to AG-CORE (by OFE-LIMIT).
- If we define $\overline{\mathcal{V}}(a) \triangleq \forall n.\ n \in \mathcal{V}(a)$, then AG-VALID and AG-VALID-STEP are equivalent. This is a natural definition because it says that an element is (fully) valid if it is valid at all step-indices.

- The rule AG-AGREE-STEP implies AG-AGREE, but the converse is not true. In fact, AG-AGREE-STEP is significantly stronger because it lets us derive "approximate" equality of x and y from "approximate" validity.

In order to fulfill these requirements, we can no longer define the composition as a case distinction on whether or not the two operands are fully equal. This case distinction completely ignores the case where the operands are approximately equal, introducing a kind of "non-continuity", which is what breaks non-expansiveness of composition in AG_0.

So instead, we use a completely different idea: An element of $\text{AG}(T)$ is a non-empty finite set of elements of T. Composition is just set union. We can think of this set as recording which elements have been composed so far. The "evaluation" of whether that composition makes any sense is "delayed" until validity checking: An element of $\text{AG}(T)$ is valid for n steps if all elements in that set are equal *up to* $(\overset{n}{=})$, *i.e.,* if the set is a singleton when viewed for just n steps. More formally, we define $\text{AG}(T)$ for any OFE T:

$$\text{AG}(T) \triangleq \left\{ a \in \wp^{\text{fin}}(T) \;\middle|\; a \neq \emptyset \right\} \qquad |a| \triangleq a$$

$$a \overset{n}{=} b \triangleq (\forall x \in a.\, \exists y \in b.\, x \overset{n}{=} y) \wedge (\forall y \in b.\, \exists x \in a.\, x \overset{n}{=} y) \qquad a \cdot b \triangleq a \cup b$$

$$\mathcal{V}(a) \triangleq \left\{ n \;\middle|\; \forall x, y \in a.\, x \overset{n}{=} y \right\} \qquad \mathsf{ag}(x) \triangleq \{x\}$$

The step-indexed equality on $\text{AG}(T)$ is defined so that we have $a \overset{n}{=} b$ if a and b contain the same elements up to the step-indexed equality $(\overset{n}{=})$ on T. This definition makes composition (defined as union) a non-expansive operation: For $a_1 \overset{n}{=} a_2$, it is easy to see that $a_1 \cdot b \overset{n}{=} a_2 \cdot b$ and $b \cdot a_1 \overset{n}{=} b \cdot a_2$. (Note: Composition would not be non-expansive if $a \overset{n}{=} b$ were instead defined as $\forall x \in a.\, \forall y \in b.\, x \overset{n}{=} y$.) The other properties (AG-CORE-STEP, AG-VALID-STEP and AG-AGREE-STEP) are also easily verified.

4.4 Cameras

The $\text{AG}(T)$ construction as defined in the previous section is no longer just an RA—we had to define $\mathcal{V}$ as a step-indexed predicate, and we had to prove non-expansiveness of the operations. In this section we introduce the algebraic structure of *cameras* to abstractly characterize such "step-indexed RAs".[9] Essentially, cameras arise by applying the translation given in §4.1 to RAs. You can find its definition in Figure 9. The differences from RAs are as follows:

- The carrier needs to be an OFE, and the core and composition operations have to be non-expansive.

[9] The reader may wonder why on earth we call them "cameras". The reason, which may not be entirely convincing, is that "camera" was originally just used as a comfortable pronunciation of "CMRA", the name used in earlier Iris papers. CMRA was originally supposed to be an acronym for "complete metric resource algebras" (or something like that), but we were never very satisfied with it and thus ended up never spelling it out. To make matters worse, the "complete" part of CMRA is now downright misleading, for whereas previously the carrier of a CMRA was required to be a COFE (complete OFE), in the present paper we relax that restriction and permit it to be an (incomplete) OFE (§8.1 goes into more detail about this change). For these reasons, we have decided to stick with the name "camera", for purposes of continuity, but to drop any pretense that it stands for something.

A *camera* is a tuple $(M : \mathbf{OFE}, \mathcal{V} : M \xrightarrow{\text{ne}} \mathit{SProp}, |-| : M \xrightarrow{\text{ne}} M^?, (\cdot) : M \times M \xrightarrow{\text{ne}} M)$ satisfying:

$$\forall a,b,c.\ (a \cdot b) \cdot c = a \cdot (b \cdot c) \qquad \text{(CAMERA-ASSOC)}$$
$$\forall a,b.\ a \cdot b = b \cdot a \qquad \text{(CAMERA-COMM)}$$
$$\forall a.\ |a| \in M \Rightarrow |a| \cdot a = a \qquad \text{(CAMERA-CORE-ID)}$$
$$\forall a.\ |a| \in M \Rightarrow ||a|| = |a| \qquad \text{(CAMERA-CORE-IDEM)}$$
$$\forall a,b.\ |a| \in M \land a \preccurlyeq b \Rightarrow |b| \in M \land |a| \preccurlyeq |b| \qquad \text{(CAMERA-CORE-MONO)}$$
$$\forall a,b.\ \mathcal{V}(a \cdot b) \subseteq \mathcal{V}(a) \qquad \text{(CAMERA-VALID-OP)}$$
$$\forall n,a,b_1,b_2.\ n \in \mathcal{V}(a) \land a \overset{n}{=} b_1 \cdot b_2 \Rightarrow \exists c_1,c_2.\ a = c_1 \cdot c_2 \land c_1 \overset{n}{=} b_1 \land c_2 \overset{n}{=} b_2 \qquad \text{(CAMERA-EXTEND)}$$

where

$$a \preccurlyeq b \triangleq \exists c.\ b = a \cdot c \qquad \text{(CAMERA-INCL)}$$
$$a \overset{n}{\preccurlyeq} b \triangleq \exists c.\ b \overset{n}{=} a \cdot c \qquad \text{(CAMERA-INCLN)}$$
$$a \rightsquigarrow B \triangleq \forall n, c^? \in M^?.\ n \in \mathcal{V}(a \cdot c^?) \Rightarrow \exists b \in B.\ n \in \mathcal{V}(b \cdot c^?)$$
$$a \rightsquigarrow b \triangleq a \rightsquigarrow \{b\}$$

Fig. 9. Camera operations and axioms.

- The predicate $\overline{\mathcal{V}}$ that defines valid elements is replaced by the step-indexed variant $\mathcal{V}$. The predicate $\mathcal{V}$ must be non-expansive (so at step-index n, it may not distinguish n-equal elements of M). Furthermore, $\mathcal{V}(a)$ being an *SProp* enforces that at smaller step-indices, only more elements can become valid. The rule CAMERA-VALID-OP expresses that we need decomposition to preserve validity at *every* step-index.
  We can recover the simpler RA validity via $\overline{\mathcal{V}}(a) \triangleq \forall n.\ n \in \mathcal{V}(a)$. RA-VALID-OP then follows from CAMERA-VALID-OP. The definition of a frame-preserving update $a \rightsquigarrow B$ is also affected by this change, and again, the camera version of a frame-preserving update implies the RA version.
  Just as the validity predicate $\overline{\mathcal{V}}$ of an RA encodes the fact that composition is partial, the step-indexed validity predicate $\mathcal{V}$ of a camera encodes "step-indexed partiality". This is more general than using a partial function, which can only be "fully defined" or "fully undefined".
- The notion of inclusion comes in two versions: The step-indexed version, $(\overset{n}{\preccurlyeq})$, and the equality-based one, $(\preccurlyeq)$. It is worth noting that, in general, these relations are reflexive only when the core $|-|$ is a total function. However, the main use of these relations as order relations appears in the definition of uniform predicates in §4.5, where we assume that the core is total[10], so this is not a problem in practice.
- The *extension* axiom CAMERA-EXTEND roughly states that decomposition must commute with step-indexed equivalence. This property is needed for proving the validity of the rule $\triangleright(P * Q) \dashv\vdash \triangleright P * \triangleright Q$. The issue is discussed in more depth in our technical appendix (Iris Team, 2017).

Notice that every RA can be trivially turned into a camera by equipping it with the discrete OFE Δ and letting $\mathcal{V}(a) \triangleq \{n \mid \overline{\mathcal{V}}(a)\}$, *i.e.,* a is either valid at all step-indices or at

[10] This follows from the fact that there exists a unit.

no step-index. Furthermore, functions between discrete OFEs are trivially non-expansive. This means that a user who does not need higher-order ghost state can safely ignore this new construction and need not prove that their functions are non-expansive.

We can easily generalize existing RA constructions to camera constructions by lifting the step-indexed validity. For example, for $+_\ell$ from §3.1, this is done as follows:

$$\mathcal{V}(a) \triangleq \{n \mid (\exists a_1 \in M_1.\, a = \mathsf{inl}(a_1) \land n \in \mathcal{V}_1(a_1)) \lor (\exists a_2 \in M_2.\, a = \mathsf{inr}(a_2) \land n \in \mathcal{V}_2(a_2))\}$$

4.5 Uniform predicates as a model of separation logic

We have now established enough infrastructure to come back to our original goal of modeling propositions of separation logic in a step-indexed fashion. In this setting, a proposition will be a step-indexed predicate over any unital camera (*i.e.,* a camera with a unit element). We call such predicates *uniform predicates*. The formal definition of the OFE of uniform predicates over some camera M is based on monotone, non-expansive *SProp* predicates:

$$M \xrightarrow{\text{mon,ne}} \mathit{SProp} \triangleq \left\{ \Phi : M \xrightarrow{\text{ne}} \mathit{SProp} \;\middle|\; \forall n, a, b.\, a \overset{n}{\preccurlyeq} b \Rightarrow \Phi(a) \overset{n}{\subseteq} \Phi(b) \right\}$$
$$\Phi \overset{n}{=} \Psi \triangleq \forall a.\, \Phi(a) \overset{n}{=} \Psi(a)$$

That is, $M \xrightarrow{\text{mon,ne}} \mathit{SProp}$ is the OFE of non-expansive step-indexed predicates that are monotone with respect to camera inclusion $\overset{n}{\preccurlyeq}$ and *SProp* inclusion $\overset{n}{\subseteq}$ *at every step-index*. This monotonicity requirement is common in models of affine separation logic.

To obtain uniform predicates, we take a quotient over the monotone step-indexed predicates defined above. The goal of the quotient is to identify predicates that only differ on invalid elements. For the purpose of the logic, we only care about valid elements—we want to equate predicates when they agree on all valid elements. Since validity is a step-indexed notion, obtaining this quotient requires not only a quotient on the carrier set but also a modified step-indexed equality ($\overset{n}{=}$):

$$\mathit{UPred}(M) \triangleq M \xrightarrow{\text{mon,ne}} \mathit{SProp}/_{\equiv}$$
$$\Phi \equiv \Psi \triangleq \forall m, a.\, m \in \mathcal{V}(a) \Rightarrow (m \in \Phi(a) \iff m \in \Psi(a))$$
$$\Phi \overset{n}{=} \Psi \triangleq \forall m \leq n, a.\, m \in \mathcal{V}(a) \Rightarrow (m \in \Phi(a) \iff m \in \Psi(a))$$

An important property of $\mathit{UPred}(M)$ is that it is *complete*. Completeness is crucial to solve the recursive domain equation (§4.6) and to model a fixed-point operator in the logic (§5.6). Notice that $\mathit{UPred}(M)$ is complete even if M is not complete. In fact, as we will discuss further in §8.1, $\mathit{UPred}(M)$ turns out to be the only OFE that we need to be complete.

Having defined *UPred* as a semantic domain of propositions, we can now define *semantic entailment* between two elements of $\mathit{UPred}(M)$: Φ entails Ψ if Ψ holds "more often" on valid elements, or formally speaking:

$$\Phi \vDash \Psi \triangleq \forall n, a.\, n \in \mathcal{V}(a) \land n \in \Phi(a) \Rightarrow n \in \Psi(a)$$

4.6 The Iris model

We now have everything set up to construct a model of Iris that supports higher-order ghost state. In order to obtain such a model, we will solve the following equation:

$$iProp \cong UPred(Res) \qquad \text{where} \qquad Res \triangleq F(iProp) \tag{IRIS}$$

This looks very much like PRE-IRIS from the beginning of this section, except that we are now using uniform predicates $UPred(Res)$ (*i.e.,* predicates involving a step-index) instead of ordinary monotone predicates $Res \xrightarrow{\text{mon}} Prop$.

We can rewrite this equation into a fixed-point problem over some G as follows:

$$G(T) \triangleq UPred(F(T))$$

It is easy to verify that a fixed-point $iProp$ of G is sufficient to satisfy IRIS.

We solve this fixed-point using America and Rutten's theorem (America & Rutten, 1989; Birkedal *et al.*, 2010). In order to state this theorem, we cannot entirely avoid category theory. Readers not familiar with category theory can simply skip the remainder of this section and jump to §4.7. Concretely, G has a fixed-point if it is a *locally contractive bifunctor* from the category of COFEs to itself. In order to understand what that means, we will define the category of COFEs and then define what it means for G to be a locally contractive bifunctor.

Categorical prerequisites of America and Rutten's theorem. The first thing that we need to define is the categories **OFE** of OFEs (respectively **COFE** of COFEs): The objects of this category are simply the OFEs (respectively COFEs), and the arrows are the non-expansive functions.

Next we need the notion of bifunctors: A bifunctor from the category **A** to the category **B** is a functor from $\mathbf{A}^{\text{op}} \times \mathbf{A}$ to **B**. Both functors and contravariant functors (*i.e.,* functors from $\mathbf{A}^{\text{op}}$ to **B**) can be considered as particular cases of bifunctors, by ignoring either the first or second argument. As a result, many OFE and COFE constructions we have seen so far can be considered as bifunctors between the corresponding categories: As examples, this is the case of $UPred$, of the non-expansive arrow $\cdot \xrightarrow{\text{ne}} \cdot$, and of $\blacktriangleright$, the later OFE.

Bifunctors can be composed: if F is a bifunctor from **B** to **C**, and G is a bifunctor from **A** to **B**, then the composition $F \circ G$ defined by $(F \circ G)(X^{\text{op}}, X) = F(G^{\text{op}}(X, X^{\text{op}}), G(X^{\text{op}}, X))$ is also a bifunctor. For simplicity's sake, we abuse notation and do not repeat the covariant and contravariant parameters. Hence, for defining $F \circ G$, we would simply have written $(F \circ G)(X) = F(G(X))$.

We now lift the notions of non-expansiveness and contractiveness from functions between OFEs to bifunctors. A bifunctor G from **COFE** to **COFE** is said to be *locally non-expansive* if its action on arrows $G : (T_2 \xrightarrow{\text{ne}} T_1) \times (T_1' \xrightarrow{\text{ne}} T_2') \to (G(T_1, T_1') \xrightarrow{\text{ne}} G(T_2, T_2'))$ is itself a non-expansive map. Here, non-expansive functions are equipped with an OFE with the usual step-indexed equality:

$$f \overset{n}{=} g \triangleq \forall u.\ f(u) \overset{n}{=} g(u)$$

In a similar spirit, a bifunctor G is *locally contractive* if its action on arrows is contractive. The functor $\blacktriangleright$ (see Figure 8) is the archetypical example of a locally contractive bifunctor.

Now we can state America and Rutten's theorem:

Theorem 2 (America and Rutten)**.** *Let* 1 *be the discrete COFE on the unit type:* $1 \triangleq \Delta\{()\}$. *Given a locally contractive bifunctor* $G : \mathbf{COFE}^{op} \times \mathbf{COFE} \to \mathbf{COFE}$*, and provided that* $G(1,1)$ *is inhabited, then there exists a unique COFE* T *such that* $G(T^{op}, T) \cong T$ (i.e., *the two are isomorphic in* **COFE**)*.*

Using America and Rutten's theorem. To apply America and Rutten's theorem, we have to show that $G(T) \triangleq \mathit{UPred}(F(T))$ is in fact a locally contractive bifunctor. G is composed from F and *UPred*, and similarly, we will decompose the proof of local contractiveness. To achieve that, we define the category **Camera** of cameras: We say that f is an arrow in the category if it is a non-expansive function and further satisfies the following properties of structure preservation (making f a camera homomorphism):

$$\forall a.\ |a| \in M \Rightarrow |f(a)| = f(|a|) \qquad \forall a.\ |a| = \bot \Rightarrow |f(a)| = \bot$$
$$\forall a, b.\ f(a \cdot b) = f(a) \cdot f(b) \qquad \forall n, a.\ n \in \mathcal{V}(a) \Rightarrow n \in \mathcal{V}(f(a))$$

Now that we have defined these categories, it is easy to show that the various constructions on these categories like *UPred*, AG, *etc.* are locally non-expansive, that $\blacktriangleright$ is locally contractive, and that composing locally non-expansive and contractive functors preserves local contractiveness.

4.7 Instantiating the Iris model

As we have seen, the model of propositions *iProp* is a solution of the equation IRIS:

$$\mathit{iProp} \cong \mathit{UPred}(\mathit{Res}) \qquad \text{where} \qquad \mathit{Res} \triangleq F(\mathit{iProp})$$

In this equation, F is a user-selected function from OFEs to unital cameras, which describes the type of ghost state one wishes to instantiate Iris with. In order to solve this equation, the function F has to satisfy some restrictions (it has to be a locally contractive bifunctor, as discussed in §4.6). However, it is important to note that the user of Iris does not have to understand what exactly these restrictions are; it is sufficient to ensure that:

1. F is composed of the usual type constructors (like functions and products), arbitrary but fixed OFEs and cameras, and OFE and camera constructions we have given in this section (like $\blacktriangleright$ and AG), and
2. the recursive occurrences of X in $F(X)$ are guarded by a $\blacktriangleright$.

In case the user needs multiple different cameras described by different functions F, we can apply the construction already described in §3.2: Given an entire family of functions $(F_i)_{i \in \mathcal{I}}$, we can instantiate Iris with the "global function" F displayed below:

$$F(X) \triangleq \prod_{i \in \mathcal{I}} \mathbb{N} \xrightarrow{\text{fin}} F_i(X)$$

It is easy to show that this construction preserves the restrictions from §4.6.

Now, if one wants to obtain a version of Iris with named propositions, one could pick $F(T) \triangleq \mathrm{AG}(\blacktriangleright(T))$. This way, we obtain ghost state of type $\mathrm{AG}(\blacktriangleright \mathit{iProp})$—*i.e.,* we can use any logical proposition and put it into a resource. The crucial reason why this is sound,

contrary to the naive version of named propositions that we have seen in §3.3, is that validity only gives an equality at a lower step-index, *i.e.,* given $n > 0$, we have:

$$n \in \mathcal{V}(\mathsf{ag}(\mathsf{next}(x)) \cdot \mathsf{ag}(\mathsf{next}(y))) \Rightarrow x \overset{n-1}{=} y$$

As we will show in the next section, this equality will be internalized in the logic as:

$$\mathcal{V}(\mathsf{ag}(\mathsf{next}(x)) \cdot \mathsf{ag}(\mathsf{next}(y))) \Rightarrow \triangleright(x = y) \qquad \text{(SPROP-AGREE)}$$

Contrary to the inconsistent rule SPROP0-AGREE from §3.3, the equality on the right-hand side of this rule is now guarded by the later modality $\triangleright$, which reflects that it holds only at the next step-index. This is a weaker conclusion than we might wish for, but it has the benefit of restoring consistency, and as we shall see, it is still quite powerful.

5 The Iris base logic

Now that we have the semantic domain *iProp* of propositions in hand, we can define the primitive connectives of the logic as elements of that domain. As already mentioned, Iris attempts to keep the base logic as small as possible. In particular, the base logic has no concept of programs and does not provide any primitive connective to reason about them—many of the connectives presented in §2 are missing in the base logic. In §6 and §7, we will show how these missing connectives can be defined—inside the logic—in terms of just the propositions of the base logic.

As is common for grammars of higher-order logic, the grammar of Iris is essentially the simply-typed lambda calculus equipped with some primitive types (of which, of course, the type iProp of logical propositions is the most interesting one) and with primitive terms operating on these types.[11] The following grammar defines both the valid types of the logic as well as its terms.[12] We use P, Q, ... as meta-variables for terms that are propositions, and t, u, ... as meta-variables for general terms of any type.

$$\tau ::= \mathsf{M} \mid \mathsf{iProp} \mid 0 \mid 1 \mid \blacktriangleright\tau \mid \tau \times \tau \mid \tau + \tau \mid \tau \to \tau \mid \ldots$$

$$\begin{aligned} t, u, P, Q ::= {} & x \mid \mathsf{next}(t) \mid \lambda x : \tau.\, t \mid t(u) \mid a \mid |t| \mid t \cdot u \mid \mu x : \tau.\, t \mid \\ & \mathsf{False} \mid \mathsf{True} \mid t =_\tau u \mid P \Rightarrow Q \mid P \wedge Q \mid P \vee Q \mid P * Q \mid P \mathrel{-\!\!*} Q \mid \\ & \exists x : \tau.\, P \mid \forall x : \tau.\, P \mid \mathsf{Own}(t) \mid \mathcal{V}(t) \mid \Box P \mid \triangleright P \mid \dot{\Rightarrow} P \mid \ldots \end{aligned}$$

We omit the usual terms for introduction (if applicable) and elimination of sums, products, unit and the empty type. We also do not show the definition of the well-typedness judgment $\Gamma \vdash t : \tau$, where Γ assigns types to free variables. It is mostly standard, the only exception being a restriction on recursive predicates: they have to be *guarded*. In $\mu x.\, t$, the variable x can only appear in t under the later ($\triangleright$) modality.

The type M is the type of the elements of the unital camera by which Iris is parameterized, and the term a lets us use any unital camera element inside the logic.

[11] Note that we write iProp for the *type* of propositions in Iris, versus *iProp* for the semantic domain used to interpret them.

[12] Note that we use similar notations $\lambda x : \tau.\, t$ and $\lambda x.\, e$ for a term of the Iris logic and a lambda-abstraction in our programming language, respectively. It should always be clear from context which of these notions is used.

The ellipses above hint at the fact that the logic may be extended with additional types and terms by the user. Formally speaking, the user picks a *signature* of types, terms and proof rules that they want to add to the logic, *e.g.,* to be able to talk about lists and reason about their properties. This is formally spelled out in the technical appendix. In the following, we assume that standard types like sets, lists, *etc.* together with their operations and properties are appropriately reflected into the logic.

The semantic interpretation function $[\![-]\!]$ for types and $[\![- : \tau]\!]_\rho$ for terms of type τ in variable assignment ρ is given in Figure 10. Notice that the interpretation of every logical connective comes with a proof obligation: namely, to verify the side conditions of *UPred*, as spelled out in §4.5.

It is important to note that functions $\tau \to \tau'$ are interpreted as non-expansive functions $[\![\tau]\!] \xrightarrow{\text{ne}} [\![\tau']\!]$. In particular, this means that the predicate Φ in the rule EQ-LEIBNIZ, which lets us substitute equals by equals, is interpreted as a non-expansive function $[\![\tau]\!] \xrightarrow{\text{ne}} \mathit{iProp}$. Non-expansiveness is crucial to justify this rule. In addition, since functions are interpreted as non-expansive functions, it is necessary to establish that the map $[\![P : \mathsf{iProp}]\!]_\rho$ is non-expansive in the variable assignment ρ.

The rules for the logical provability judgment[13] $P \vdash Q$ are displayed in Figure 11. Here, $P \dashv\vdash Q$ is shorthand for having both $P \vdash Q$ and $Q \vdash P$. The connection between the definition of the model and the logic is made by the following key theorem:

Theorem 3 (Soundness of the semantic model)**.**

$$(P \vdash Q) \Rightarrow [\![P]\!] \vDash [\![Q]\!]$$

This theorem says that if a judgment can be derived using the rules, then the semantic interpretations of the involved propositions are related by entailment in the semantic model (as defined in §4.5).

We will now go over the proof rules and model of the basic connectives in more detail. We do not discuss the ordinary connectives of intuitionistic higher-order logic, which are standard both in terms of their model (given in Figure 10) and their rules (displayed in the technical appendix). The remaining connectives and proof principles fall into two broad categories: those dealing with ownership of resources (§5.1-§5.4) and those related to step-indexing (§5.5-§5.7).

5.1 Separation logic

The connectives $*$ and $-\!\!*$ of bunched implications (O'Hearn & Pym, 1999) make our base logic a *separation logic*: they let us reason about *ownership of resources*. The key point is that $P * Q$ describes ownership of a resource that can be *separated* into two disjoint pieces, one satisfying P and one satisfying Q. This is in contrast to $P \wedge Q$, which describes ownership of a resource satisfying both P and Q.

For example, consider the resources owned by different threads in a concurrent program. Because these threads operate concurrently, it is crucial that their ownership is *disjoint*. As

[13] The full judgment is of the shape $\Gamma \mid P \vdash Q$. However, since Γ only plays a role in the rules for quantifiers, we omit it.

$$\llbracket \mathsf{iProp} \rrbracket \triangleq \mathit{iProp} \qquad \llbracket \blacktriangleright \tau \rrbracket \triangleq \blacktriangleright \llbracket \tau \rrbracket$$
$$\llbracket \mathsf{M} \rrbracket \triangleq \mathit{Res} \qquad \llbracket \tau_1 \times \tau_2 \rrbracket \triangleq \llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket$$
$$\llbracket 1 \rrbracket \triangleq \Delta\{()\} \qquad \llbracket \tau \to \sigma \rrbracket \triangleq \llbracket \tau \rrbracket \xrightarrow{\mathrm{ne}} \llbracket \sigma \rrbracket$$

$$\llbracket \mathsf{False} : \mathsf{iProp} \rrbracket_\rho \triangleq \lambda_.\, \emptyset$$
$$\llbracket \mathsf{True} : \mathsf{iProp} \rrbracket_\rho \triangleq \lambda_.\, \mathbb{N}$$
$$\llbracket t =_\tau u : \mathsf{iProp} \rrbracket_\rho \triangleq \lambda_.\, \left\{ n \;\middle|\; \llbracket t : \tau \rrbracket_\rho \overset{n}{=} \llbracket u : \tau \rrbracket_\rho \right\}$$
$$\llbracket P \wedge Q : \mathsf{iProp} \rrbracket_\rho \triangleq \lambda a.\, \llbracket P : \mathsf{iProp} \rrbracket_\rho(a) \cap \llbracket Q : \mathsf{iProp} \rrbracket_\rho(a)$$
$$\llbracket P \vee Q : \mathsf{iProp} \rrbracket_\rho \triangleq \lambda a.\, \llbracket P : \mathsf{iProp} \rrbracket_\rho(a) \cup \llbracket Q : \mathsf{iProp} \rrbracket_\rho(a)$$
$$\llbracket P \Rightarrow Q : \mathsf{iProp} \rrbracket_\rho \triangleq \lambda a.\, \left\{ n \;\middle|\; \begin{array}{l} \forall m \le n, b.\, a \preccurlyeq b \wedge m \in \mathcal{V}(b) \Rightarrow \\ \qquad m \in \llbracket P : \mathsf{iProp} \rrbracket_\rho(b) \Rightarrow m \in \llbracket Q : \mathsf{iProp} \rrbracket_\rho(b) \end{array} \right\}$$
$$\llbracket \forall x : \tau.\, P : \mathsf{iProp} \rrbracket_\rho \triangleq \lambda a.\, \bigcap_{v \in \llbracket \tau \rrbracket} \llbracket P : \mathsf{iProp} \rrbracket_{\rho[x \leftarrow v]}(a)$$
$$\llbracket \exists x : \tau.\, P : \mathsf{iProp} \rrbracket_\rho \triangleq \lambda a.\, \bigcup_{v \in \llbracket \tau \rrbracket} \llbracket P : \mathsf{iProp} \rrbracket_{\rho[x \leftarrow v]}(a)$$
$$\llbracket P * Q : \mathsf{iProp} \rrbracket_\rho \triangleq \lambda a.\, \left\{ n \;\middle|\; \exists b_1, b_2.\, a \overset{n}{=} b_1 \cdot b_2 \wedge n \in \llbracket P : \mathsf{iProp} \rrbracket_\rho(b_1) \wedge n \in \llbracket Q : \mathsf{iProp} \rrbracket_\rho(b_2) \right\}$$
$$\llbracket P -\!\!* Q : \mathsf{iProp} \rrbracket_\rho \triangleq \lambda a.\, \left\{ n \;\middle|\; \begin{array}{l} \forall m \le n, b.\, m \in \mathcal{V}(a \cdot b) \Rightarrow \\ \qquad m \in \llbracket P : \mathsf{iProp} \rrbracket_\rho(b) \Rightarrow m \in \llbracket Q : \mathsf{iProp} \rrbracket_\rho(a \cdot b) \end{array} \right\}$$
$$\llbracket \mathsf{Own}(t) : \mathsf{iProp} \rrbracket_\rho \triangleq \lambda b.\, \left\{ n \;\middle|\; \llbracket t : \mathsf{M} \rrbracket_\rho \overset{n}{\preccurlyeq} b \right\}$$
$$\llbracket \mathcal{V}(t) : \mathsf{iProp} \rrbracket_\rho \triangleq \lambda_.\, \mathcal{V}(\llbracket t : \mathsf{M} \rrbracket_\rho)$$
$$\llbracket \Box P : \mathsf{iProp} \rrbracket_\rho \triangleq \lambda a.\, \llbracket P : \mathsf{iProp} \rrbracket_\rho(|a|)$$
$$\llbracket \dot{\Rightarrow} P : \mathsf{iProp} \rrbracket_\rho \triangleq \lambda a.\, \left\{ n \;\middle|\; \begin{array}{l} \forall m \le n, c.\, m \in \mathcal{V}(a \cdot c) \Rightarrow \\ \qquad \exists b.\, m \in \mathcal{V}(b \cdot c) \wedge m \in \llbracket P : \mathsf{iProp} \rrbracket_\rho(b) \end{array} \right\}$$
$$\llbracket \mu x.\, t : \mathsf{iProp} \rrbracket_\rho \triangleq \mathit{fix}_{\llbracket \tau \rrbracket}(\lambda u.\, \llbracket t : \mathsf{iProp} \rrbracket_{\rho[x \leftarrow u]})$$
$$\llbracket \triangleright P : \mathsf{iProp} \rrbracket_\rho \triangleq \lambda a.\, \left\{ n \;\middle|\; n = 0 \vee n - 1 \in \llbracket P : \mathsf{iProp} \rrbracket_\rho(a) \right\}$$

$$\llbracket x : \tau \rrbracket_\rho \triangleq \rho(x) \qquad \llbracket (t_1, t_2) : \tau_1 \times \tau_2 \rrbracket_\rho \triangleq (\llbracket t_1 : \tau_1 \rrbracket_\rho, \llbracket t_2 : \tau_2 \rrbracket_\rho)$$
$$\llbracket \lambda x : \tau.\, t : \tau \to \sigma \rrbracket_\rho \triangleq \lambda u.\, \llbracket t : \sigma \rrbracket_{\rho[x \leftarrow u]} \qquad \llbracket \pi_i(t) : \tau_i \rrbracket_\rho \triangleq \pi_i(\llbracket t : \tau_1 \times \tau_2 \rrbracket_\rho)$$
$$\llbracket t(u) : \sigma \rrbracket_\rho \triangleq \llbracket t : \tau \to \sigma \rrbracket_\rho(\llbracket u : \tau \rrbracket_\rho) \qquad \llbracket a : \mathsf{M} \rrbracket_\rho \triangleq a$$
$$\llbracket () : 1 \rrbracket_\rho \triangleq () \qquad \llbracket |t| : \mathsf{M} \rrbracket_\rho \triangleq |\llbracket t : \mathsf{M} \rrbracket_\rho|$$
$$\llbracket \mathsf{next}(t) : \blacktriangleright \tau \rrbracket_\rho \triangleq \mathsf{next}(\llbracket t : \tau \rrbracket_\rho) \qquad \llbracket t \cdot u : \mathsf{M} \rrbracket_\rho \triangleq \llbracket t : \mathsf{M} \rrbracket_\rho \cdot \llbracket u : \mathsf{M} \rrbracket_\rho$$

Fig. 10. Semantic interpretation of the Iris base logic.

Laws for equality.

EQ-REFL
$\mathsf{True} \vdash x =_\tau x$

EQ-LEIBNIZ
$x =_\tau y \vdash \Phi(x) \Rightarrow \Phi(y)$

EQ-NEXT
$\mathsf{next}(x) =_{\blacktriangleright\tau} \mathsf{next}(y) \dashv\vdash \triangleright(x =_\tau y)$

Laws of (affine) bunched implications.

$$\mathsf{True} * P \dashv\vdash P \qquad P * Q \vdash Q * P \qquad (P * Q) * R \vdash P * (Q * R)$$

$*$-MONO
$$\frac{P_1 \vdash Q_1 \qquad P_2 \vdash Q_2}{P_1 * P_2 \vdash Q_1 * Q_2}$$

$-\!\!*$-INTRO
$$\frac{P * Q \vdash R}{P \vdash Q -\!\!* R}$$

$-\!\!*$-ELIM
$$\frac{P \vdash Q -\!\!* R}{P * Q \vdash R}$$

Laws for resources and validity.

OWN-OP
$\mathsf{Own}(a) * \mathsf{Own}(b) \dashv\vdash \mathsf{Own}(a \cdot b)$

OWN-UNIT
$\mathsf{True} \vdash \mathsf{Own}(\varepsilon)$

OWN-CORE
$\mathsf{Own}(a) \vdash \Box\mathsf{Own}(|a|)$

OWN-VALID
$\mathsf{Own}(a) \vdash \mathcal{V}(a)$

VALID-OP
$\mathcal{V}(a \cdot b) \vdash \mathcal{V}(a)$

VALID-PERSISTENT
$\mathcal{V}(a) \vdash \Box\mathcal{V}(a)$

Laws for the basic update modality.

UPD-MONO
$$\frac{P \vdash Q}{\Rrightarrow P \vdash \Rrightarrow Q}$$

UPD-INTRO
$P \vdash \Rrightarrow P$

UPD-TRANS
$\Rrightarrow\Rrightarrow P \vdash \Rrightarrow P$

UPD-FRAME
$Q * \Rrightarrow P \vdash \Rrightarrow(Q * P)$

UPD-UPDATE
$$\frac{a \rightsquigarrow B}{\mathsf{Own}(a) \vdash \Rrightarrow \exists b \in B.\ \mathsf{Own}(b)}$$

Laws for the persistence modality.

$\Box$-MONO
$$\frac{P \vdash Q}{\Box P \vdash \Box Q}$$

$\Box$-IDEMP
$\Box P \vdash \Box\Box P$

$\Box$-ELIM
$\Box P \vdash P$

$\Box$-CONJ
$\Box P \wedge Q \vdash \Box P * Q$

$\forall x.\ \Box P \vdash \Box\forall x.\ P$
$\Box\exists x.\ P \vdash \exists x.\ \Box P$

Laws for the later modality.

$\triangleright$-MONO
$$\frac{P \vdash Q}{\triangleright P \vdash \triangleright Q}$$

LÖB
$(\triangleright P \Rightarrow P) \vdash P$

$\triangleright(P * Q) \dashv\vdash \triangleright P * \triangleright Q$
$\Box\triangleright P \dashv\vdash \triangleright\Box P$

$\forall x.\ \triangleright P \vdash \triangleright\forall x.\ P$
$\triangleright\exists x.\ P \vdash \triangleright\mathsf{False} \vee \exists x.\ \triangleright P$

Laws for timeless propositions.

$\triangleright$-TIMELESS
$\triangleright P \vdash \triangleright\mathsf{False} \vee (\triangleright\mathsf{False} \Rightarrow P)$

$\triangleright$-OWN
$\triangleright\mathsf{Own}(a) \vdash \exists b.\ \mathsf{Own}(b) \wedge \triangleright(a =_{\mathsf{M}} b)$

Fig. 11. Proof rules of the Iris base logic.

a consequence, separating conjunction is the natural operator to combine the ownership of concurrent threads.

Notice, however, that the base logic does not have a notion of "threads" or "programs", but only of *separation*. Ultimately, the logic does not care about "who" owns the resources under consideration; it is only concerned with reasoning about consequences of ownership and how it can be merged and split disjointly. The connection to threads will be made when we encode Hoare triples in the base logic (see §6.3 and §7.3).

Together with separating conjunction, we have a second form of implication: the *magic wand* $P -\!\!* Q$. It describes ownership of "Q minus P", *i.e.,* it describes resources such that, if you (disjointly) add resources satisfying P, you obtain resources satisfying Q.

This is in contrast to implication $P \Rightarrow Q$, where instead of adding a disjoint resource that satisfies P, it is demanded that the same resource additionally (in the sense of a "normal" conjunction) satisfies P. In other words, to eliminate an implication, we use the rule $P \wedge (P \Rightarrow Q) \vdash Q$, but to eliminate a magic wand, we need to demand the separating conjunction, as in $P * (P \mathrel{-\!\!*} Q) \vdash Q$.

The model of separating conjunction is the usual one for intuitionistic separation logic (Reynolds, 2000), just slightly adapted to the step-indexed setting. As usual in a Kripke semantics (Kripke, 1965), both implication and magic wand have to be closed under smaller step-indices and bigger resources.

5.2 Resource ownership

The purpose of the $\mathsf{Own}(a)$ connective is to assert ownership of the resource $a : \mathsf{M}$. Recall that M is interpreted by a unital camera *Res*, the resources by which Iris is parameterized, so that the user can decide what kind of ghost ownership they wish to have.

Since *Res* is a unital camera, we can use the step-indexed inclusion $\overset{n}{\preccurlyeq}$ (CAMERA-INCLN) to define the model of $\mathsf{Own}(a)$ as "the current resources are at least a":

$$[\![\mathsf{Own}(t) : \mathsf{iProp}]\!]_\rho \triangleq \lambda b.\ \{n \mid [\![t : \mathsf{M}]\!]_\rho \overset{n}{\preccurlyeq} b\}$$

Notice that the resources should be at least a and not exactly a; this is what makes our logic affine, see also §9.5.

The proposition $\mathsf{Own}(a)$ forms the "primitive" form of ownership in our logic, which can then be composed into more interesting propositions using the previously described connectives. The most important fact about ownership is that separating conjunction "reflects" the composition operator of RAs into the logic (OWN-OP).

Besides the $\mathsf{Own}(a)$ connective, we have the primitive connective $\mathcal{V}(a)$, which reflects the step-indexed validity of camera elements into the logic. Note that ownership is connected to validity: the rule OWN-VALID says that only valid elements can be owned.

5.3 The persistence modality and persistent resources

In §2.3, we explained that some propositions are persistent, meaning that they hold forever and that they can be arbitrarily duplicated. For example, validity, equality, and ownership of a core are all persistent propositions. This notion of being persistent is expressed in Iris by means of the *persistence* modality $\Box$.

The purpose of the *persistence* modality $\Box P$ is to say that P holds *without asserting any exclusive ownership*. This is modeled by defining $\Box P$ to hold for some resource a if P holds for $|a|$, the core of a:[14]

$$[\![\Box P : \mathsf{iProp}]\!]_\rho \triangleq \lambda a.\ [\![P : \mathsf{iProp}]\!]_\rho(|a|)$$

As a consequence, the assumption $\Box P$ can be used arbitrarily often, *i.e.,* cannot be "used up"—it is persistent. We can now *define* what it means for a proposition to be persistent:

$$\mathsf{persistent}(P) \triangleq P \vdash \Box P$$

[14] Remember that *Res* has a unit, and hence the $|-|$ is a total function.

Using just the rules in Figure 11, it is easy to verify that validity $\mathcal{V}(a)$, equality $t = u$, ownership of a core $\mathsf{Own}(|a|)$, and the persistence modality $\Box P$ itself are persistent. Furthermore, it is easy to verify that any persistent proposition P enjoys: $P * P \Leftrightarrow P \Leftrightarrow \Box P$ and $P * Q \Leftrightarrow P \wedge Q$.

The persistence modality gives more flexibility than just having a notion of persistent propositions: we make use of this flexibility to define view shifts (§7.2) and Hoare triples (§6) in such a way that they are persistent. For now, we will give a simple teaser.

Consider the proposition $P \mathrel{-\!\!*} Q$. This proposition can generally only be applied (to produce the result Q) *once*, because its proof may depend on some exclusively owned resource. However, we can use the persistence modality to "tag" that the proof of $P \mathrel{-\!\!*} Q$ does not depend on any exclusive ownership. This gives rise to $\Box(P \mathrel{-\!\!*} Q)$, which is a persistent "fact" that can be duplicated and applied repeatedly.

Obviously, this comes at the price that when proving $\Box(P \mathrel{-\!\!*} Q)$, we cannot make use of any non-persistent hypotheses in the surrounding context. This is witnessed by the following derived rule for $\Box$ introduction, which only allows one to introduce a persistence modality $\Box$ whenever the context is persistent:

$$\frac{\Box P \vdash Q}{\Box P \vdash \Box Q}\ \Box\text{-INTRO}$$

In fact, this is exactly the usual introduction rule for the **!** of linear logic. (For Iris, we picked the equivalent formulation using $\Box$-MONO and $\Box$-IDEMP instead, but both are equivalent.) In other words, the persistence modality is a lot like the **!**, except that it supports some more operations ($\Box$-CONJ and commuting with quantifiers).

5.4 The basic update modality

So far, resources have been *static*: the base logic provides propositions to reason about resources you own, the consequences of that ownership, and how ownership can be disjointly separated. What has been missing is a way to *change* the current resources. As discussed in §2.1, the natural notion of a resource update is the *frame-preserving update*. The purpose of the *(basic) update modality* $\dot{\Rightarrow} P$ is to reflect frame-preserving updates into the logic. The proposition $\dot{\Rightarrow} P$ thus holds for some resource a, if from a we can do a frame-preserving update to some b that satisfies P:

$$[\![\dot{\Rightarrow} P : \mathsf{iProp}]\!]_\rho \triangleq \lambda a. \left\{ n \;\middle|\; \begin{array}{l} \forall m \le n, c.\ m \in \mathcal{V}(a \cdot c) \Rightarrow \\ \qquad \exists b.\ m \in \mathcal{V}(b \cdot c) \wedge m \in [\![P : \mathsf{iProp}]\!]_\rho(b) \end{array} \right\}$$

The update modality $\dot{\Rightarrow} P$ provides a way, inside the logic, to talk about the resources we *could own* after performing an update to what we *do* own.

The rule UPD-UPDATE witnesses the relationship between the $\dot{\Rightarrow}$ modality and frame-preserving updates, while the remaining proof rules of $\dot{\Rightarrow}$ essentially say that $\dot{\Rightarrow}$ is a *strong monad* with respect to separating conjunction (Kock, 1970; Kock, 1972).

This gives rise to an alternative interpretation of the basic update modality: we can think of $\dot{\Rightarrow} P$ as a *thunk* (or "suspended computation") that captures some resources in its environment and that, when executed, will "return" resources satisfying P. The various

proof rules then let us perform additional reasoning on the result of the thunk (UPD-MONO), create a thunk that does nothing (UPD-INTRO), compose two thunks into one (UPD-TRANS), and add resources to those captured by the thunk (UPD-FRAME).

In §2, we introduced *view shifts* $P \Rrightarrow_{\mathcal{E}} Q$ as the "ghost moves" of Iris. View shifts and the update modality are clearly closely related: both describe updates of the ghost state. In fact, in §7.2, we will see how to *encode* view shifts in terms of the update modality. We have not yet introduced enough machinery to give the full definition, but we give here a teaser with the following simplified definition, the *basic view shift*:

$$P \dot{\Rrightarrow} Q \triangleq \Box(P \mathrel{-\!\!*} \dot{\Rrightarrow} Q)$$

View shifts have a precondition P imposing some restrictions on when they can be applied; this is expressed using the magic wand. The update modality is inserted in the conclusion in order to allow updates in the ghost state to happen when the basic view shift is "executed". Finally, we use the persistence modality $\Box$ to make view shifts persistent. It is easy to show that the basic view shift enjoys most of the rules given in Figure 4, except those involving invariants.

5.5 The later modality

As we have seen in §4, Iris uses *step-indexing* to obtain a solution to the recursive domain equation defining the semantic domain of propositions. This step-indexing aspect of the model is internalized into the logic by adding the *later modality*, $\triangleright P$ (Nakano, 2000; Appel *et al.*, 2007). The modality is modeled as follows: $\triangleright P$ holds for some resource at some step-index if P holds for the same resource *at the next (lower) step-index*. In the definition of Hoare triples in §6, we will connect $\triangleright$ to steps of the program, allowing one to think of $\triangleright P$ as saying that P holds at the "next step of computation".

One interesting aspect of working with a step-indexed model is that propositions like $x = y$ and $\mathcal{V}(a)$ are *implicitly* referring to the current step index. For example, AG-AGREE-STEP could be expressed inside Iris as $\forall x, y.\ \mathcal{V}(\mathsf{ag}(x) \cdot \mathsf{ag}(y)) \Rightarrow x = y$. Combined with the rule EQ-NEXT in Figure 11, we can then derive SPROP-AGREE, as we already hinted at in the previous section:

$$\mathcal{V}(\mathsf{ag}(\mathsf{next}(x)) \cdot \mathsf{ag}(\mathsf{next}(y))) \Rightarrow \triangleright(x = y)$$

It is important to understand that this statement does *not* just talk about the case where $\mathsf{ag}(\mathsf{next}(x)) \cdot \mathsf{ag}(\mathsf{next}(y))$ is "fully" valid (*i.e.,* valid at all step-indices); rather, it asserts that the validity of this camera element for n steps implies that $x \overset{n-1}{=} y$.

Note that many of the usual rules for later, such as those displayed below, are derivable from the rules in Figure 11.

$$\triangleright\text{-INTRO}\quad P \vdash \triangleright P \qquad \triangleright\text{-AND}\quad \triangleright(P \land Q) \dashv\vdash \triangleright P \land \triangleright Q \qquad \triangleright\text{-EXIST}\quad \frac{\mathsf{inhabited}(x)}{\triangleright(\exists x.\ P) \dashv\vdash \exists x.\ \triangleright P}$$

5.6 Guarded fixed-points

Step-indexing not only gives us a way to build a model of Iris propositions. It is also a powerful tool for defining propositions recursively: under some conditions, we can define

propositions as being the unique fixed-point of some functions over propositions. In the logic, this is internalized by the *guarded fixed-point* operator $\mu x.\, t$. This fixed-point operator can be used to define recursive predicates *without any restriction on the variance* of the recursive occurrences of x in t. Instead, all recursive occurrences of x must be *guarded*: they have to appear below a later modality $\triangleright$.

In §6, we will show how guarded recursion is used for defining weakest preconditions. Moreover, as shown in Svendsen & Birkedal (2014), guarded recursion is useful for defining specifications for higher-order concurrent data structures.

In the model, the existence of such a unique fixed-point is guaranteed by Banach's fixed-point theorem:

Theorem 4 (Banach's fixed-point)**.** *Given an inhabited COFE T and a contractive function $f : T \to T$, there exists a unique fixed-point $\mathit{fix}_T f$ such that $f(\mathit{fix}_T f) = \mathit{fix}_T f$.*

We have already seen in §4.5 that $\mathit{iProp} = \mathit{UPred}(\mathit{Res})$ is complete. The syntactic guardedness condition turns out to be sufficient to ensure the contractiveness of the predicate. The reason for this is that the interpretation of $\triangleright$ is contractive, while the interpretations of all the other logical connectives are non-expansive, as we have seen earlier in §5. One can easily verify that composing a contractive function with a non-expansive one preserves contractiveness.

A crucial proof rule for $\triangleright$ is LÖB, which allows assuming that the current goal holds *later*. The proof of the LÖB rule proceeds by performing induction over the step-index. LÖB facilitates proving properties about fixed-points: we can essentially assume that the recursive occurrences (which are below a later) are already proven correct. We will see LÖB induction in action in §6, where we define Hoare triples using a guarded fixed-point.

5.7 Timeless propositions

There are some occasions where we inevitably end up with hypotheses below a later. The prime example is the rule for accessing invariants (HOARE-INV in §2). Although one can always introduce a later ($\triangleright$-INTRO), one cannot just eliminate a later, so the later may make certain reasoning steps impossible.

As we will prove in §8.2, it is crucial for logical soundness that when opening *impredicative* invariants P, we get access to P below a later modality. However, if P does not refer to other invariants, it is generally safe to get P immediately, *i.e.,* without a later modality. In the Iris program logic, this is facilitated using the following rule:

$$\frac{\text{VS-TIMELESS}\quad \mathsf{timeless}(P)}{\triangleright P \Rrightarrow_{\mathcal{E}} P}$$

The intuitive meaning of $\mathsf{timeless}(P)$ (pronounced "P is *timeless*") is that P is a first-order predicate that does not refer to other invariants. When we compose this rule with HOARE-INV and HOARE-VS, we can derive the following rule for opening invariants without a later:

$$\frac{\text{HOARE-INV-TIMELESS}\quad \{P * Q_1\}\, e\, \{v.\, P * Q_2\}_{\mathcal{E}\setminus\mathcal{N}} \qquad \mathsf{timeless}(P) \qquad \mathsf{atomic}(e) \qquad \mathcal{N} \subseteq \mathcal{E}}{\left\{\boxed{P}^{\mathcal{N}} * Q_1\right\} e \left\{v.\, \boxed{P}^{\mathcal{N}} * Q_2\right\}_{\mathcal{E}}}$$

So, what does it mean for a proposition to be timeless? Semantically speaking, a proposition P is timeless if, whenever it holds at step-index 0, it also holds at all step-indices. More precisely, as usual, we restrict attention to step-indices at which the resource under consideration is valid:

$$\forall n, a.\, n \in \mathcal{V}(a) \;\wedge\; 0 \in [\![P]\!](a) \;\Rightarrow\; n \in [\![P]\!](a)$$

With this definition in hand, we could prove suitable rules for timeless propositions in the model of the logic. This is what we have done previously in Iris 1.0 and Iris 2.0.

In Iris 3.x we proceed in a different way: we wish to define what it means for a proposition to be timeless *within the logic*. Naively, one might expect a proposition P to be timeless if it satisfies $\triangleright P \vdash P$. However, in the presence of LÖB, this entailment would give us that $P = \mathsf{True}$! The reason for this is that $\triangleright P$ is always true at step-index 0. To define timelessness, we thus have to exclude the step-index 0.

Excluding the step-index 0 is achieved using the "except 0" modality $\diamond$, as defined below, whose semantics is that the given proposition holds at all step-indices greater than 0. This makes use of the fact that $\triangleright \mathsf{False}$ holds only at step-index 0. Using this modality, we then define timelessness in the logic as follows:

$$\diamond P \triangleq P \vee \triangleright \mathsf{False}$$
$$\mathsf{timeless}(P) \triangleq \triangleright P \vdash \diamond P$$

Using just the rules in Figure 11, we can prove that most of the connectives of our logic preserve timelessness: when P and Q are timeless, the propositions $P \wedge Q$, $P \vee Q$, $P \Rightarrow Q$, $\forall x.\, P$, $\exists x.\, P$, $P * Q$, $P \mathrel{-\!\!*} Q$, and $\Box P$ are timeless. Most of these proofs are straightforward, but for $P \Rightarrow Q$, $P \mathrel{-\!\!*} Q$, and $\forall x.\, P$ we have to make use of the rule ▷-TIMELESS. This somewhat strange-looking rule allows us to essentially do case analysis on whether the step-index is 0 or not. Indeed, the proposition $\triangleright \mathsf{False} \vee (\triangleright \mathsf{False} \Rightarrow P)$ holds precisely when either the step-index is zero (the left disjunct), or P holds at step-index zero (the right disjunct). Thus for timeless propositions (which hold at all step-indices if they hold at zero), we can read the whole rule as saying that if $\triangleright P$ holds then either the step index is zero, or P holds.

Note that even if P is timeless, then $\triangleright P$ generally is not. In fact, Löb induction shows that if $\triangleright P$ is timeless then $\mathsf{True} \vdash \triangleright P$.

Furthermore, we can show that if a camera M is *discrete—i.e.,* it degenerates to a plain RA with a discrete equality and discrete validity—then the propositions for equality $a = b$ and validity $\mathcal{V}(a)$ of elements in $a, b : M$ are timeless. Then, using the rule ▷-OWN we can derive that ownership $\boxed{a : M}^{\gamma}$ of elements of a discrete camera M is timeless.

These properties make timeless propositions a powerful notion. The cameras of many commonly appearing types of resources are discrete—for example, our oneshot RA and the traditional separation-logic RA of heaps with disjoint union. To this end, when we put just first-order propositions involving ownership of discrete resources in invariants, we can open said invariants without a later. Thus, using the notion of timelessness we can recover some of the convenience of logics which support only first-order, predicative invariants, while retaining the ability to form and use higher-order invariants when desired.

It is important to notice that the $\diamond$ modality "except 0" is useful beyond defining what it means for a proposition to be timeless. Its most important property is that when we have

a goal that is behind the $\diamond$ modality, then we can strip laters off of timeless hypotheses, as shown by the derived rule $\diamond$-TIMELESS below.

$$\frac{P \vdash Q}{\diamond P \vdash \diamond Q}\ \diamond\text{-MONO} \qquad P \vdash \diamond P\ \diamond\text{-ELIM} \qquad \diamond\diamond P \vdash \diamond P\ \diamond\text{-ELIM} \qquad \frac{P \vdash \diamond Q \qquad \mathsf{timeless}(P)}{\triangleright P \vdash \diamond Q}\ \diamond\text{-TIMELESS}$$

In §7.2 we will see that the $\diamond$ modality can be used to define the view shift in the base logic, and it will allow us to derive the rule VS-TIMELESS from the primitive rules of the base logic.

5.8 Consistency

Logical consistency is usually stated as $\mathsf{True} \not\vdash \mathsf{False}$: from a closed context one cannot prove a contradiction. This theorem holds for Iris, but we will strengthen it in two ways.

- If we unfold this statement slightly, we arrive at $(\mathsf{True} \vdash \mathsf{False}) \Rightarrow \mathsf{False}$ where the implication and the second False are on the meta-level. In some sense, this theorem says that we can "move" proofs of False performed inside the logic in the empty context to the "outside world". If we want to use our logic for more than just stating provability (notably, we want to prove a Hoare triple and show that its postcondition actually reflects the program behavior, as in §7.4), we will need a stronger statement where we can move more interesting propositions to the "outside".
- Furthermore, we have to consider the fact that Iris has various modalities. Certainly, $\mathsf{True} \not\vdash \Box\mathsf{False}$ is covered by the above statement (because the persistence modality can just be eliminated, $\Box$-ELIM), but what about $\mathsf{True} \not\vdash \triangleright\mathsf{False}$ or $\mathsf{True} \not\vdash \Rightarrow\!\!\!\!\!\!\cdot\ \mathsf{False}$? We certainly want these statements to hold, but they do not follow from the usual notion of consistency as shown above.

These considerations lead us to the following theorem:

Theorem 5 (Soundness of first-order interpretation)**.** *Given a first-order proposition ϕ (not involving ownership, higher-order quantification, or any of the modalities) and $\mathsf{True} \vdash (\Rightarrow\!\!\!\!\!\!\cdot\ \triangleright)^n\ \phi$, then the "standard" (meta-logic) interpretation of ϕ holds. Here, $(\Rightarrow\!\!\!\!\!\!\cdot\ \triangleright)^n$ is notation for nesting n times the modalities $\Rightarrow\!\!\!\!\!\!\cdot$ and $\triangleright$.*

The proposition ϕ should be a first-order predicate to ensure it can be used equivalently both inside our logic and at the meta-level (*i.e.,* it enjoys $\phi \iff \forall a, n.\ n \in [\![\phi]\!](a)$). Given that, Theorem 5 is a straightforward consequence of the soundness of the model (Theorem 3). The usual notion of consistency, *i.e.,* $\mathsf{True} \not\vdash \mathsf{False}$, follows trivially from Theorem 5: just pick $\phi = \mathsf{False}$ and $n = 0$.

Furthermore, while our soundness theorem is restricted to alternating sequences of $\triangleright$ and $\Rightarrow\!\!\!\!\!\!\cdot$, it actually implies consistency under a combination of modalities: We already discussed that the persistence modality can just be removed. Beyond this, "unevenly" nested occurrences of $\triangleright$ and $\Rightarrow\!\!\!\!\!\!\cdot$ can easily be brought into the shape $(\Rightarrow\!\!\!\!\!\!\cdot\ \triangleright)^n$ by collapsing adjacent occurrences of $\Rightarrow\!\!\!\!\!\!\cdot$ (UPD-TRANS) and inserting $\Rightarrow\!\!\!\!\!\!\cdot$ between adjacent occurrences of $\triangleright$ (UPD-INTRO).

6 Weakest preconditions

As a warm-up for §7, where we will describe the encoding of the entire Iris program logic in terms of the Iris base logic, we first show in this section how to encode a stripped-down concurrent program logic. This stripped-down program logic features Hoare triples, but it does not yet support (impredicative) invariants. For this reason, we will temporarily ignore the masks annotating Hoare triples (and the weakest precondition modality, described below). Furthermore, we will for simplicity use the ML-like programming language from §2, whereas the actual Iris program logic can be instantiated with an arbitrary programming language.

Usually, program logics are centered around Hoare triples. However, instead of directly defining Hoare triples in the Iris base logic, we first define the notion of a *weakest precondition*. Using weakest preconditions, Hoare triples can then be defined as:

$$\{P\}\, e\, \{\Phi\} \triangleq \Box(P \mathrel{-\!\!*} \mathsf{wp}\, e\, \{\Phi\}).$$

Notice that Hoare triples relate to weakest preconditions just as basic view shifts (§5.4) relate to basic updates: The restriction enforced by the precondition P is expressed using the magic wand, and the persistence modality $\Box$ is used to make the Hoare triple persistent.

Given a postcondition $\Phi : \mathit{Val} \to \mathsf{iProp}$, the connective $\mathsf{wp}\ e\ \{\Phi\}$ gives the *weakest precondition* under which all executions of e are *safe* and all return values v satisfy $\Phi(v)$. For an execution to be safe, we demand that it *does not get stuck*. In the case of our ML-like programming language, this means, in particular, that the program must never access invalid locations and that, for all $\texttt{assert}(e)$ statements, if e terminates, it should evaluate to $\texttt{true}$.

In §6.1, we first define the operational semantics of the programming language we introduced in §2. In §6.2, we then give some intuitions about weakest preconditions and how to work with them. After that, in §6.3, we present the encoding of weakest preconditions in three stages, gradually adding support for reasoning about state and concurrency. Finally, in §6.4 we prove adequacy of weakest preconditions.

6.1 Operational semantics

Figure 12 shows the operational semantics of the ML-like language that we introduced in §2 and will consider in this section. For clarity's sake, we repeat the syntax below:

$$\begin{aligned}
v \in \mathit{Val} &::= () \mid z \mid \texttt{true} \mid \texttt{false} \mid \ell \mid \lambda x.e \mid \ldots && (z \in \mathbb{Z})\\
e \in \mathit{Expr} &::= v \mid x \mid e_1(e_2) \mid \texttt{fork}\ \{e\} \mid \texttt{assert}(e) \mid\\
&\qquad \texttt{ref}(e) \mid\, !e \mid e_1 \leftarrow e_2 \mid \texttt{CAS}(e,e_1,e_2) \mid \ldots\\
K \in \mathit{Ctx} &::= \bullet \mid K(e) \mid v(K) \mid \texttt{assert}(K) \mid \texttt{ref}(K) \mid\, !K \mid K \leftarrow e \mid v \leftarrow K \mid\\
&\qquad \texttt{CAS}(K,e_1,e_2) \mid \texttt{CAS}(v,K,e_2) \mid \texttt{CAS}(v,v_1,K) \mid \ldots
\end{aligned}$$

Head reduction $(e,\sigma) \to_{\mathsf{h}} (e',\sigma',\vec{e}_f)$ is defined on pairs (e,σ) consisting of an expression e and a shared heap σ (a finite partial map from locations to values). Moreover, $\vec{e}_f$ is a list of forked-off expressions, which is used to define the semantics of $\texttt{fork}\ \{e\}$. The head-reduction is lifted to a per-thread reduction $(e,\sigma) \to_{\mathsf{t}} (e',\sigma',\vec{e}_f)$ using evaluation contexts. We define an expression e to be reducible in a shared heap σ, written $\mathrm{red}(e,\sigma)$, if it can

Head reduction.

$$
\begin{array}{rll}
((\lambda x.e)(v),\sigma) \to_{\mathsf{h}} (e[v/x],\sigma,\varepsilon) & & \\
(\texttt{fork}\ \{e\},\sigma) \to_{\mathsf{h}} ((),\sigma,e) & & \\
(\texttt{assert}(\texttt{true}),\sigma) \to_{\mathsf{h}} ((),\sigma,\varepsilon) & & \\
(\texttt{ref}(v),\sigma) \to_{\mathsf{h}} (\ell,\sigma[\ell \leftarrow v],\varepsilon) & & \text{if } \sigma(\ell) = \bot \\
(!\ell,\sigma) \to_{\mathsf{h}} (v,\sigma,\varepsilon) & & \text{if } \sigma(\ell) = v \\
(\ell \leftarrow w,\sigma) \to_{\mathsf{h}} ((),\sigma[\ell \leftarrow w],\varepsilon) & & \text{if } \sigma(\ell) = v \\
(\texttt{CAS}(\ell,v,w),\sigma) \to_{\mathsf{h}} (\texttt{true},\sigma[\ell \leftarrow w],\varepsilon) & & \text{if } \sigma(\ell) = v \\
(\texttt{CAS}(\ell,v,w),\sigma) \to_{\mathsf{h}} (\texttt{false},\sigma,\varepsilon) & & \text{if } \sigma(\ell) \neq \bot \text{ and } \sigma(\ell) \neq v
\end{array}
$$

Thread-local and threadpool reduction.

$$\frac{(e,\sigma) \to_{\mathsf{h}} (e',\sigma',\vec{e}_f)}{(K[e],\sigma) \to_{\mathsf{t}} (K[e'],\sigma',\vec{e}_f)} \qquad \frac{(e,\sigma) \to_{\mathsf{t}} (e',\sigma',\vec{e}_f)}{(T_1;e;T_2,\sigma) \to_{\mathsf{tp}} (T_1;e';T_2;\vec{e}_f,\sigma')}$$

Fig. 12. Operational semantics.

make a thread-local step. The threadpool reduction $(T,\sigma) \to_{\mathsf{tp}} (T',\sigma')$ is an interleaving semantics where the threadpool T denotes the existing threads as a list of expressions.

Note that in this language, compare-and-set can be used with any type of values, including sums and products, which is unrealistic. This is for simplicity only: Other instantiations of Iris use a more realistic compare-and-set semantics (Jung *et al.*, 2018).

6.2 Proof rules

In order to give some more intuition about weakest preconditions, we discuss some of the proof rules and indicate how these can be used to prove the Hoare rules in Figure 1. Those already familiar with weakest preconditions can proceed to §6.3.

Figure 13 shows the most important rules of the $\mathsf{wp}\ e\ \{\Phi\}$ connective. These rules are inspired by the "backwards" style Hoare rules of Ishtiaq & O'Hearn (2001), but presented in weakest-precondition style. As usual in a weakest-precondition style system (Dijkstra, 1975), the postcondition of the conclusion of each rule involves an arbitrary predicate Φ.

For example, imagine we are given $P \vdash Q$ and want to prove:

$$\ell \mapsto v * P \vdash \mathsf{wp}\ (\ell \leftarrow w)\ \{r.\ r = () * \ell \mapsto w * Q\},$$

which is equivalent to the Hoare triple $\{\ell \mapsto v * P\}\ \ell \leftarrow w\ \{r.\ r = () * \ell \mapsto w * Q\}$. The rule WP-STORE tells us which resources we have to supply and which resources we get in return. This becomes evident in the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{P \vdash Q}{P * \ell \mapsto w \vdash () = () * \ell \mapsto w * Q}\ *\text{-MONO}}{P \vdash \ell \mapsto w \mathrel{-\!\!*} () = () * \ell \mapsto w * Q}\ \mathrel{-\!\!*}\text{-INTRO}}{\ell \mapsto v * P \vdash \ell \mapsto v * \triangleright(\ell \mapsto w \mathrel{-\!\!*} () = () * \ell \mapsto w * Q)}\ *\text{-MONO}, \triangleright\text{-INTRO}}{\ell \mapsto v * P \vdash \mathsf{wp}\ (\ell \leftarrow w)\ \{r.\ r = () * \ell \mapsto w * Q\}}\ \text{WP-STORE}$$

Here, we use $*$-MONO to prove that we own the location ℓ—this should not be surprising; in a separation logic, we have to demonstrate ownership of a location to access it. Then, using our remaining resources P we have to prove $\ell \mapsto w \mathrel{-\!\!*} () = () * \ell \mapsto w * Q$. We do this by introducing the wand, which gives us the location ℓ with its updated value.

Notice the end-to-end effect of applying this little derivation: we gave up $\ell \mapsto v$, and it got replaced in our context with $\ell \mapsto w$. However, this was all expressed in the *premise* of WP-STORE (and similarly for the other rules), with the conclusion applying to an *arbitrary* postcondition Φ.

Keeping the above derivation in mind, it is straightforward to derive the rules for Hoare triples as shown in Figure 1. Take HOARE-STORE for example: By expanding the definition of the Hoare triple, we end up with $\square(\ell \mapsto v \mathrel{-\!\!*} \mathsf{wp}\ (\ell \leftarrow w)\ \{\ell \mapsto w\})$, which can be proven using roughly the same derivation as shown above.

Note that the rules for weakest preconditions in Figure 13 are in fact stronger than those for Hoare triples in Figure 1 due to the presence of the later in the premise, which allows one to strip off a later of any hypothesis in the surrounding context. This functionality is crucial when one uses Löb induction to reason about fixed-point computations—it enables hypotheses introduced by Löb induction (under a later) to become usable (by having the later stripped off) after the program has taken some physical step of computation.

It may be surprising that there is no frame rule in Figure 13. It turns out that WP-WAND implies framing. In fact, WP-WAND is equivalent to the following two more conventional rules:

$$\text{WP-MONO}\quad \frac{\forall v.\ \Phi(v) \vdash \Psi(v)}{\mathsf{wp}\ e\ \{\Phi\} \vdash \mathsf{wp}\ e\ \{\Psi\}} \qquad \text{WP-FRAME}\quad P * \mathsf{wp}\ e\ \{\Phi\} \vdash \mathsf{wp}\ e\ \{P * \Phi\}$$

It is useful to note that weakest preconditions are more convenient for performing interactive proofs with Iris (Krebbers *et al.*, 2017b). In order to use the rules for Hoare triples (*e.g.,* those for primitive instructions, like HOARE-LOAD and HOARE-STORE) one first needs to bring the pre- and postcondition into a specific shape, which is done via framing and the rule of consequence. In contrast, the rules for weakest preconditions (*e.g.,* WP-LOAD and WP-STORE) are applicable with any postcondition and implicitly have framing and consequence built in, which allows for much easier interactive proofs, as we have seen in the example proof above.

6.3 Definition of weakest preconditions

We now discuss how to *define* weakest preconditions using the Iris base logic, proceeding in three stages of increasing complexity. As part of defining weakest preconditions, we will also have to define the points-to connective.

6.3.1 First stage

To get started, let us assume the program we want to verify makes no use of fork or shared heap access. The idea of $\mathsf{wp}\ e\ \{\Phi\}$ is to ensure that given any reduction $(e, \sigma) \to_{\mathsf{t}} \cdots \to_{\mathsf{t}} (e_n, \sigma_n)$, either (e_n, σ_n) is reducible, or the program terminated, *i.e.,* e_n is a value v for which we have $\Phi(v)$. The natural candidate for encoding this is using the fixed-point operator

$$(\forall v.\ \Phi(v) \mathrel{-\!\!*} \Psi(v)) * \mathsf{wp}\ e\ \{\Phi\} \vdash \mathsf{wp}\ e\ \{\Psi\} \qquad \text{(WP-WAND)}$$
$$\Phi(v) \vdash \mathsf{wp}\ v\ \{\Phi\} \qquad \text{(WP-VAL)}$$
$$\mathsf{wp}\ e\ \{v.\ \mathsf{wp}\ K[v]\ \{\Phi\}\} \vdash \mathsf{wp}\ K[e]\ \{\Phi\} \qquad \text{(WP-BIND)}$$
$$\triangleright\Phi() * \triangleright \mathsf{wp}\ e\ \{v.\ \mathsf{True}\} \vdash \mathsf{wp}\ \texttt{fork}\ \{e\}\ \{\Phi\} \qquad \text{(WP-FORK)}$$
$$\triangleright\Phi() \vdash \mathsf{wp}\ \texttt{assert}(\texttt{true})\ \{\Phi\} \qquad \text{(WP-ASSERT)}$$
$$\triangleright \mathsf{wp}\ e[v/x]\ \{\Phi\} \vdash \mathsf{wp}\ (\lambda x.e)v\ \{\Phi\} \qquad \text{(WP-}\lambda\text{)}$$
$$\triangleright(\forall \ell.\ \ell \mapsto v \mathrel{-\!\!*} \Phi(\ell)) \vdash \mathsf{wp}\ \texttt{ref}(v)\ \{\Phi\} \qquad \text{(WP-ALLOC)}$$
$$\ell \mapsto v * \triangleright(\ell \mapsto v \mathrel{-\!\!*} \Phi(v)) \vdash \mathsf{wp}\ !\ell\ \{\Phi\} \qquad \text{(WP-LOAD)}$$
$$\ell \mapsto v * \triangleright(\ell \mapsto w \mathrel{-\!\!*} \Phi()) \vdash \mathsf{wp}\ (\ell \leftarrow w)\ \{\Phi\} \qquad \text{(WP-STORE)}$$
$$\ell \mapsto v * \triangleright(\ell \mapsto w \mathrel{-\!\!*} \Phi(\texttt{true})) \vdash \mathsf{wp}\ \texttt{CAS}(\ell, v, w)\ \{\Phi\} \qquad \text{(WP-CAS-SUC)}$$
$$v \neq v' \land \ell \mapsto v * \triangleright(\ell \mapsto v \mathrel{-\!\!*} \Phi(\texttt{false})) \vdash \mathsf{wp}\ \texttt{CAS}(\ell, v', w)\ \{\Phi\} \qquad \text{(WP-CAS-FAIL)}$$

Fig. 13. Rules for weakest preconditions.

$\mu x.\, t$ of our logic. Consider the following:

$$\begin{aligned} \mathsf{wp}\ e\ \{\Phi\} \triangleq\ & (e \in \mathit{Val} \land \Phi(e)) && \textit{(return value)} \\ & \lor \big(e \notin \mathit{Val} \land \forall \sigma.\ \mathrm{red}(e, \sigma) && \textit{(safety)} \\ & \quad \land \triangleright \forall e_2, \sigma_2.\ \big((e, \sigma) \to_{\mathrm{t}} (e_2, \sigma_2, \varepsilon)\big) \mathrel{-\!\!*} \mathsf{wp}\ e_2\ \{\Phi\} && \textit{(preservation)} \quad \big) \end{aligned}$$

Weakest precondition is defined by case distinction: Either the program has already terminated (e is a value), in which case the postcondition should hold, or the program is *not* a value, in which case there are two requirements. First, for any possible heap σ, the program should be reducible (called *program safety*). Second, *if* the program makes a step, then the weakest precondition of the reduced program e_2 must hold (called *preservation*).

Note that the recursive occurrence $\mathsf{wp}\ e_2\ \{\Phi\}$ appears under a $\triangleright$-modality, so the above can indeed be defined using the fixed-point operator μ. In some sense, this "ties" the steps of the program to the step-indices implicit in the logic, by adding another $\triangleright$ for every program step.

So, how useful is this definition? The rules WP-VAL and WP-λ are almost trivial, and using LÖB induction we can prove WP-WAND and WP-BIND. We can thus reason about programs that do not fork or make use of the heap.

But unfortunately, this definition cannot be used to verify programs involving heap accesses: the states σ and σ_2 are universally quantified and not related to anything. The program must always be able to proceed under *any* heap, so we cannot possibly prove the rules of the load, store, and allocation constructs.

The usual way to proceed in constructing a separation logic is to define the pre- and postconditions as *predicates* over states, but that is not the direction we take. After all, our base logic *already* has a notion of "resources that can be updated"—*i.e.,* a notion of state—built into its model of propositions. Of course we want to make use of this power in building our program logic.

6.3.2 Second stage: Adding state

We now consider programs that access the shared heap but still do not fork. To use the resources provided by the Iris base logic, we have to start by thinking about the right camera to encode the points-to connective $\ell \mapsto v$ and to connect it to the current shared heap. An obvious candidate would be to use $\mathit{Loc} \xrightarrow{\text{fin}} \text{Ex}(\mathit{Val})$[15] and define $\ell \mapsto v$ as $\boxed{[\ell \leftarrow \mathsf{ex}(v)]}^{\gamma_{\text{HEAP}}}$, where γ_{HEAP} is a fixed name of a ghost location. However, that leaves us with a problem: how do we tie the resources $\ell \mapsto v$ to the actual heap that the program executes on? We have to make sure that from *owning* $\ell \mapsto v$ we can actually deduce that the location ℓ is allocated in the heap σ and has value v.

To this end, we will actually have *two* heaps in our resources, *both* consisting of elements of $\mathit{Loc} \xrightarrow{\text{fin}} \text{Ex}(\mathit{Val})$. The *authoritative heap* $\bullet\sigma$ is managed by the weakest precondition and tied to the physical state occurring in the program reduction. There will only ever be one authoritative heap resource, *i.e.,* we want $\bullet\sigma \cdot \bullet\sigma'$ to be invalid. At the same time, the *heap fragments* $\circ\sigma$ will be owned by the program itself and used to give meaning to $\ell \mapsto v$. Specifically, $\ell \mapsto v$ will be encoded as $\boxed{\circ[\ell \leftarrow v]}^{\gamma_{\text{HEAP}}}$, *i.e.,* as ownership of the singleton heap fragment governing ℓ. These fragments can be composed the usual way, *i.e.,* $\circ\sigma \cdot \circ\sigma' = \circ(\sigma \uplus \sigma')$. Finally, we need to tie these two pieces together, making sure that the fragments are always a "part" of the authoritative state: if $\bullet\sigma \cdot \circ\sigma'$ is valid, then $\sigma' \preccurlyeq \sigma$ should hold.

The above reasoning pattern can be formalized as an instance, $\text{Auth}(\mathit{Loc} \xrightarrow{\text{fin}} \text{Ex}(\mathit{Val}))$, of a more general Iris construction, called the *authoritative camera*, which we present in §6.3.3. Before we explain how to define the authoritative camera, let us see why it is useful in the definition of weakest preconditions. The new definition is (changes are in blue):

$$
\begin{aligned}
&\mathsf{wp}\, e\, \{\Phi\} \triangleq (e \in \mathit{Val} \wedge \dot{\Rightarrow} \Phi(e)) \\
&\qquad \vee \Big(e \notin \mathit{Val} \wedge \forall \sigma.\ \boxed{\bullet\sigma}^{\gamma_{\text{HEAP}}} \mathrel{-\!\!*} \dot{\Rightarrow} \big(\text{red}(e,\sigma) \\
&\qquad\quad \wedge \triangleright \forall e_2, \sigma_2.\ \big((e,\sigma) \to_{\mathrm{t}} (e_2,\sigma_2,\varepsilon)\big) \mathrel{-\!\!*} \dot{\Rightarrow} \big(\boxed{\bullet\sigma_2}^{\gamma_{\text{HEAP}}} * \mathsf{wp}\, e_2\, \{\Phi\}\big)\big)\Big) \\
&\ell \mapsto v \triangleq \boxed{\circ[\ell \leftarrow v]}^{\gamma_{\text{HEAP}}}
\end{aligned}
$$

The difference from the first definition is that the second disjunct (the one covering the case of a program that can still reduce) requires proving safety and preservation under the assumption that the authoritative heap $\bullet\sigma$ *matches the physical one*. Moreover, when the program makes a step to some new state σ_2, the proof must be able to *produce* a matching authoritative heap. Finally, the basic update modality permits the proof to perform frame-preserving updates.

To see why this is useful, consider proving WP-LOAD, the weakest precondition of the operation $!\ell$. After picking the right disjunct of the definition of the weakest precondition and introducing all assumptions, we can combine the assumptions made by the rule, $\ell \mapsto v$, with the assumptions provided by the definition of weakest preconditions to obtain $\boxed{\bullet\sigma \cdot \circ[\ell \leftarrow v]}^{\gamma_{\text{HEAP}}}$. By OWN-VALID, we learn that this camera element is valid, which (as discussed above) implies $[\ell \leftarrow v] \preccurlyeq \sigma$, so $\sigma(\ell) = v$. In other words, because the camera ties

[15] This camera is isomorphic to finite partial functions with composition being disjoint union.

the authoritative heap and the heap fragments together, and weakest precondition ties the authoritative heap and the physical heap used in program reduction together, we can make a connection between $\ell \mapsto v$ and the physical heap. This is yet another example of "fictional separation": Weakest precondition treats the heap as a single unsplittable entity, but through the use of the authoritative camera we can nevertheless hand out logical ownership of pieces of the state to different parts of a program.

Completing the proof of safety and preservation is now straightforward. Since all possible reductions of $!\ell$ do not change the heap, we can produce the authoritative heap $\bullet\,\sigma_2$ by just "forwarding" the one we got earlier in the proof. In this case, we did not even make use of the fact that we are allowed to perform frame-preserving updates. Such updates are necessary, however, to prove weakest preconditions of operations that actually *change* the state (like allocation $\texttt{ref}(e)$ or storing $e_1 \leftarrow e_2$), because in those cases the authoritative heap needs to be changed likewise.

6.3.3 Authoritative camera

To complete the definition, we need to define the *authoritative camera* (Jung *et al.*, 2015), and we will do so in a general way, because there is nothing about this construction that is specific to ownership of heaps vs. any other kind of resource. Assume we are given some unital camera M and let:

$$\begin{aligned}
\textsc{Auth}(M) &\triangleq \textsc{Ex}(M)^? \times M \\
\mathcal{V}(x,b) &\triangleq \left\{ n \;\middle|\; (x = \bot \land n \in \mathcal{V}(b)) \lor (\exists a.\, x = \mathsf{ex}(a) \land n \in \mathcal{V}(a) \land b \overset{n}{\preccurlyeq} a) \right\} \\
(x_1,b_1) \cdot (x_2,b_2) &\triangleq (x_1 \cdot x_2, b_1 \cdot b_2) \\
|(x,b)| &\triangleq (\bot, |b|)
\end{aligned}$$

With $a \in M$, we write $\bullet a$ for $(\mathsf{ex}(a), \varepsilon)$ to denote *authoritative* ownership of a, and $\circ a$ for $(\bot, a)$ to denote *fragmentary* ownership of a.

It can be easily verified that this camera has the three key properties discussed above: ownership of $\bullet a$ is exclusive, ownership of $\circ a$ composes like that of a, and the two are tied together in the sense that validity of $\bullet a \cdot \circ b$ implies $b \preccurlyeq a$. In addition, as we will show in §7.1.1, we can use the authoritative camera $\mathsf{Auth}(M)$ with different choices of M, and depending on the choice of M we will get different frame-preserving updates for $\mathsf{Auth}(M)$. In particular, choosing M to be $\mathit{Loc} \xrightarrow{\text{fin}} \textsc{Ex}(\mathit{Val})$, as we have done above, we can show the following frame-preserving updates that are needed for WP-STORE and WP-ALLOC:

$$\begin{aligned}
\bullet\,\sigma \cdot \circ[\ell \leftarrow v] &\rightsquigarrow \bullet\,\sigma[\ell \leftarrow w] \cdot \circ[\ell \leftarrow w] \\
\bullet\,\sigma &\rightsquigarrow \bullet\,\sigma[\ell \leftarrow w] \cdot \circ[\ell \leftarrow w] \qquad \text{if } \ell \notin \mathrm{dom}(\sigma)
\end{aligned}$$

In our appendix (Iris Team, 2017) and the Coq formalization, we provide a generic mechanism for deriving such updates in a composable fashion.

6.3.4 Third stage: Adding fork

Our previous definition of $\mathsf{wp}\ e\ \{\Phi\}$ only talked about reductions $(e,\sigma) \to_{\mathsf{t}} (e_2,\sigma_2,\varepsilon)$ which do *not* fork off threads, and hence one could not prove WP-FORK. The following new

definition lifts this limitation:

$$\begin{aligned}\mathsf{wp}\, e\, \{\Phi\} \triangleq\ & (e \in \mathit{Val} \wedge \Rightarrow\!\!\!\!\!\!\cdot\ \Phi(e)) \\ & \vee \Big(e \notin \mathit{Val} \wedge \forall \sigma.\ \boxed{\bullet \sigma}^{\gamma_{\text{HEAP}}} \mathrel{-\!\!*} \Rightarrow\!\!\!\!\!\!\cdot\ \big(\mathrm{red}(e,\sigma) \\ & \quad \wedge \triangleright \forall e_2, \sigma_2, \vec{e}_f.\ \big((e,\sigma) \to_{\mathsf{t}} (e_2, \sigma_2, \vec{e}_f)\big) \mathrel{-\!\!*} \Rightarrow\!\!\!\!\!\!\cdot \\ & \qquad \big(\boxed{\bullet \sigma_2}^{\gamma_{\text{HEAP}}} * \mathsf{wp}\, e_2\, \{\Phi\} * \mathop{\Large *}_{e' \in \vec{e}_f} \mathsf{wp}\, e'\, \{v.\, \mathsf{True}\}\big)\big)\Big) \\ \ell \mapsto v \triangleq\ & \boxed{\circ [\ell \leftarrow v]}^{\gamma_{\text{HEAP}}}\end{aligned}$$

Instead of just demanding a proof of the weakest precondition of the thread e under consideration, we also demand proofs that all the forked-off threads $\vec{e}_f$ are safe, by using the iterated separated conjunction $*$. We do not care about their return values, so the postcondition is trivial.

This encoding shows how much mileage we get out of building on top of the Iris base logic. Because said logic supports ownership and step-indexing, we can get around explicitly managing resources and step-indices in the weakest precondition definition. We do not have to explicitly account for the way resources are subdivided between the current thread and the forked-off thread. Instead, all we have to do is surgically place some update modalities, a single $\triangleright$, and some standard separation logic connectives. This keeps the definition of, and the reasoning about, weakest preconditions nice and compact.

6.3.5 *Eliminating basic view shifts*

An important property of Hoare triples in Iris is that we can perform updates to ghost state around them. In the Iris program logic, this is realized via the rule HOARE-VS, which allows one to eliminate view shifts in the pre- and postcondition. For the definition of Hoare triples and weakest preconditions in this section, we cannot prove that rule yet. However, we can already prove a similar rule for basic view shifts $P \Rightarrow Q \triangleq \Box(P \mathrel{-\!\!*} \Rightarrow\!\!\!\!\!\!\cdot\ Q)$ (see §5.4). This rule and the corresponding rule for weakest preconditions are as follows:

$$\frac{P \Rightarrow P' \qquad \{P'\}\, e\, \{v.\, Q'\} \qquad \forall v.\, Q' \Rightarrow Q}{\{P\}\, e\, \{v.\, Q\}} \qquad\qquad \frac{\Rightarrow\!\!\!\!\!\!\cdot\ \mathsf{wp}\, e\, \{v.\, \Rightarrow\!\!\!\!\!\!\cdot\ \Phi(v)\}}{\mathsf{wp}\, e\, \{\Phi\}}$$

Recall that we have defined $\{P\}\, e\, \{\Phi\} \triangleq \Box(P \mathrel{-\!\!*} \mathsf{wp}\, e\, \{\Phi\})$. Hence, the rule on the left trivially follows from the rule on the right. Given the definitions of weakest preconditions in §6.3.2 and §6.3.4, the rule on the right can be proven using LÖB. Note that for these rules to hold, it is crucial to put an update modality $\Rightarrow\!\!\!\!\!\!\cdot$ in the value case of the definition of weakest preconditions.

6.4 *Adequacy*

To demonstrate that our weakest preconditions actually make the expected statements about program executions, we prove the following *adequacy theorem*.

Theorem 6 (Adequacy of weakest preconditions)**.** *Let ϕ be a first-order predicate. If* $\mathit{True} \vdash \mathit{wp}\, e\, \{\phi\}$ *and* $(e, \sigma) \to_{\mathsf{tp}}^{*} (e'_1 \ldots e'_n, \sigma')$*, then:*

1. *For any i with* $1 \leq i \leq n$ *we have that either* e_i' *is a value, or* $\mathrm{red}(e_i', \sigma')$*;*
2. *If* e_1' *(the main thread) is a value v, then* $\phi(v)$*.*

Proof. In order to prove this theorem, we first show that reduction steps preserve weakest preconditions. Since our language has a threadpool semantics, this means we have to show that reduction steps preserve the weakest preconditions of all threads. So, for any reduction sequence $(e_1, e_2 \ldots e_n, \sigma) \to_{\mathsf{tp}}^k (e_1', e_2' \ldots e_m', \sigma')$ of k steps, we prove:

$$\frac{\boxed{\bullet\sigma}^{\gamma_{\mathrm{HEAP}}} * \mathsf{wp}\ e_1\ \{\phi\} * \mathop{\Large *}_{2 \leq i \leq n} \mathsf{wp}\ e_i\ \{\mathsf{True}\}}{(\dot{\Rightarrow}\triangleright)^k \left(\boxed{\bullet\sigma'}^{\gamma_{\mathrm{HEAP}}} * \mathsf{wp}\ e_1'\ \{\phi\} * \mathop{\Large *}_{2 \leq j \leq m} \mathsf{wp}\ e_j'\ \{\mathsf{True}\}\right)}$$

This result is proven entirely in the Iris base logic. The proof proceeds by induction on k and unfolding the definition of weakest preconditions in each step of the induction proof.

With the above result in hand, it is now easy to show that given any reduction sequence $(e, \sigma) \to_{\mathsf{tp}}^k (e_1' \ldots e_n', \sigma')$ of k steps, we have:

$$\boxed{\bullet\sigma}^{\gamma_{\mathrm{HEAP}}} * \mathsf{wp}\ e\ \{\phi\} \vdash (\dot{\Rightarrow}\triangleright)^{k+1}\ \left(e_i' \in \mathit{Val} \lor \mathrm{red}(e_i', \sigma')\right) \qquad \text{for } 1 \leq i \leq n$$

$$\boxed{\bullet\sigma}^{\gamma_{\mathrm{HEAP}}} * \mathsf{wp}\ e\ \{\phi\} \vdash (\dot{\Rightarrow}\triangleright)^{k+1}\ \phi(e_1') \qquad \text{if } e_1' \in \mathit{Val}$$

Both proofs are done within the base logic, again, and proceed by unfolding the definition of weakest preconditions once more. Now, if we combine these results with the hypothesis $\mathsf{True} \vdash \mathsf{wp}\ e\ \{\phi\}$ and with the rule GHOST-ALLOC for allocating the ghost variable $\boxed{\bullet\sigma}^{\gamma_{\mathrm{HEAP}}}$, we get $\mathsf{True} \vdash (\dot{\Rightarrow}\triangleright)^{k+1}\ (e_i' \in \mathit{Val} \lor \mathrm{red}(e_i', \sigma'))$ and $\mathsf{True} \vdash (\dot{\Rightarrow}\triangleright)^{k+1}\ (e_1' \in \mathit{Val} \Rightarrow \phi(e_1'))$.

Since $(e_i' \in \mathit{Val} \lor \mathrm{red}(e_i', \sigma'))$ and $(e_1' \in \mathit{Val} \Rightarrow \phi(e_1'))$ are first-order propositions, and thus can be used both inside the base logic and at the meta level, we can apply Theorem 5, the soundness result of the base logic. This concludes the proof. □

7 Recovering the Iris program logic

In this section, we will show how to encode the reasoning principles of the *full* Iris program logic (Jung *et al.*, 2015; Jung *et al.*, 2016) within the base logic. Most importantly, we will derive the core proof rule for accessing invariants, HOARE-INV (§2.2), which allows opening an invariant around an atomic expression:

$$\frac{\{\triangleright P * Q_1\}\ e\ \{v.\ \triangleright P * Q_2\}_{\mathcal{E} \setminus \mathcal{N}} \qquad \mathsf{atomic(e)} \qquad \mathcal{N} \subseteq \mathcal{E}}{\left\{\boxed{P}^{\mathcal{N}} * Q_1\right\}\ e\ \left\{v.\ \boxed{P}^{\mathcal{N}} * Q_2\right\}_{\mathcal{E}}}$$

One can also present this rule directly for weakest preconditions. The following weakest precondition rule can easily be shown to imply the above rule:

$$\frac{\text{WP-INV}\quad \triangleright P \vdash \mathsf{wp}_{\mathcal{E} \setminus \mathcal{N}}\ e\ \{v.\ \triangleright P * \Phi(v)\} \qquad \mathsf{atomic(e)} \qquad \mathcal{N} \subseteq \mathcal{E}}{\boxed{P}^{\mathcal{N}} \vdash \mathsf{wp}_{\mathcal{E}}\ e\ \{\Phi\}}$$

Before proving these rules, we begin in §7.1 by establishing the required infrastructure to encode invariants. In §7.2, we then define the *fancy update modality*, which enriches the basic update modality with support for accessing invariants and removing laters ($\triangleright$) from timeless propositions. The fancy update modality is used to recover the notion of

Ghost variables/cameras

$$\begin{array}{lll} \gamma_{\text{INV}} & \text{with elements of} & \text{INV} \triangleq \text{AUTH}(\mathbb{N} \xrightarrow{\text{fin}} \text{AG}(\blacktriangleright \mathsf{iProp})) \\ \gamma_{\text{EN}} & \text{with elements of} & \text{EN} \triangleq \wp(\mathbb{N}) \mid \lightning \\ \gamma_{\text{DIS}} & \text{with elements of} & \text{DIS} \triangleq \wp^{\text{fin}}(\mathbb{N}) \mid \lightning \end{array}$$

Definitions

$$W \triangleq \exists I : \mathbb{N} \xrightarrow{\text{fin}} \mathsf{iProp}.\ \boxed{\bullet \mathsf{ag}(\mathsf{next}(I))}^{\gamma_{\text{INV}}} * \ast_{\iota \in \mathrm{dom}(I)} \left((\triangleright I(\iota) * \boxed{\{\iota\}}^{\gamma_{\text{DIS}}}) \vee \boxed{\{\iota\}}^{\gamma_{\text{EN}}} \right)$$

(Above, ag and next are implicitly mapped pointwise over I).

$$\boxed{P}^{\iota} \triangleq \boxed{\circ\, [\iota \leftarrow \mathsf{ag}(\mathsf{next}(P))]}^{\gamma_{\text{INV}}}$$

$$\boxed{P}^{\mathcal{N}} \triangleq \exists \iota \in \mathcal{N}^{\uparrow}.\ \boxed{P}^{\iota}$$

$$\mathcal{E}_1 \Rrightarrow^{\mathcal{E}_2} P \triangleq W * \boxed{\mathcal{E}_1}^{\gamma_{\text{EN}}} \mathrel{-\!\!*} \Rrightarrow^{\dot{\diamond}} (W * \boxed{\mathcal{E}_2}^{\gamma_{\text{EN}}} * P)$$

$$P \ {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} Q \triangleq \Box(P \mathrel{-\!\!*} {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} Q)$$

$$\begin{aligned} \mathsf{wp}^S_{\mathcal{E}}\, e\, \{\Phi\} \triangleq\ & (e \in \mathit{Val} \wedge \Rrightarrow_{\mathcal{E}} \Phi(e)) \\ & \vee \Big(e \notin \mathit{Val} \wedge \forall \sigma.\ S(\sigma) \mathrel{-\!\!*} {}^{\mathcal{E}}\!\Rrightarrow^{\emptyset} \left(\mathrm{red}(e,\sigma)\right. \\ & \quad \wedge \triangleright \forall e_2, \sigma_2, \vec{e}_f.\ \left((e,\sigma) \to_{\mathfrak{t}} (e_2, \sigma_2, \vec{e}_f)\right) \mathrel{-\!\!*} {}^{\emptyset}\!\Rrightarrow^{\mathcal{E}} \\ & \qquad \left. \left(S(\sigma_2) * \mathsf{wp}^S_{\mathcal{E}}\, e_2\, \{\Phi\} * \ast_{e' \in \vec{e}_f} \mathsf{wp}^S_{\top}\, e'\, \{v.\mathsf{True}\}\right)\right) \Big) \end{aligned}$$

$$\{P\}\, e\, \{\Phi\}^S_{\mathcal{E}} \triangleq \Box(P \mathrel{-\!\!*} \mathsf{wp}^S_{\mathcal{E}}\, e\, \{\Phi\})$$

Fig. 14. The Iris program logic.

view shifts. In §7.3, we proceed by giving the full definition of weakest preconditions, which extends the definition in §6.3.4 with support for opening invariants and is moreover parameterized by the language of program expressions. In particular, we prove WP-INV. We conclude this section with a generic adequacy result for the full Iris program logic in §7.4.

All definitions of the Iris program logic are shown in Figure 14. We explain these in detail in the following sections.

7.1 Invariants

This section describes the protocol governing the allocation of and access to invariants, which we call *world satisfaction*. The name "world satisfaction" originates in earlier work on Kripke logical-relations models of higher-order stateful languages, *e.g.,* (Dreyer *et al.*, 2010); there, it denoted the global invariant that the physical state (*i.e.,* the heap) should, after every step of computation, respect whatever invariants had thus far been established in a logical-relations proof, which were codified in a "possible world" parameter of the logical relation. In the context of Iris, world satisfaction means something similar, but it is generalized to a property over arbitrary resources (not just physical ones) and expressed directly as a formula in the Iris base logic, which we will end up enforcing as a global invariant simply by threading it through the definition of view shifts and weakest preconditions (see §7.2).

In §7.1.1, we define world satisfaction. However, instead of using namespaces $\mathcal{N}$ to identify invariants (as WP-INV does), world satisfaction uses a simpler mechanism: Each invariant has a dynamically chosen name $\iota \in \mathbb{N}$. In §7.1.2, we show the limitations of dynamically chosen names and introduce the concept of namespaces to overcome these limitations.

7.1.1 World satisfaction

The key idea for defining invariants in the Iris base logic is to define a single global invariant W, called *world satisfaction*, which keeps track of *all* existing invariants. The definition of world satisfaction is given below:

$$W \triangleq \exists I : \mathbb{N} \to \mathsf{iProp}.\ \boxed{\bullet \mathsf{ag}(\mathsf{next}(I))}^{\gamma_{\text{INV}}} \ast \mathop{\ast}_{\iota \in \mathrm{dom}(I)} \left((\triangleright I(\iota) \ast \boxed{\{\iota\}}^{\gamma_{\text{DIS}}}) \vee \boxed{\{\iota\}}^{\gamma_{\text{EN}}} \right)$$

$$\boxed{P}^{\iota} \triangleq \boxed{\circ [\iota \leftarrow \mathsf{ag}(\mathsf{next}(P))]}^{\gamma_{\text{INV}}}$$

Here, $I : \mathbb{N} \to \mathsf{iProp}$ is a registry of all existing invariants. The iterated separating conjunction $\ast$ makes sure that invariants that have not been opened do in fact hold. We use two tokens ($\boxed{\{\iota\}}^{\gamma_{\text{DIS}}}$ and $\boxed{\{\iota\}}^{\gamma_{\text{EN}}}$) to keep track of whether invariants have been opened or not; this mechanism will be explained in more detail below.

World satisfaction not only ensures that the existing invariants hold, but it also ensures that these invariants are tied to the occurrences of the connective $\boxed{P}^{\iota}$, which witness that P has been registered as an invariant named ι in I. We achieve this using a mechanism similar to the one tying the heap σ to the points-to connective $\ell \mapsto v$ in §6.3.2: namely, the authoritative camera. For the heap we used the camera $\textsc{Auth}(\mathit{Loc} \xrightarrow{\text{fin}} \textsc{Ex}(\mathit{Val}))$, but for invariants we will be using $\textsc{Auth}(\mathbb{N} \xrightarrow{\text{fin}} \textsc{Ag}(\blacktriangleright \mathsf{iProp}))$. These cameras differ in the following key respects:

- Rather than storing values, the map we are maintaining here is storing propositions (of type iProp). This is achieved using higher-order ghost state, which means that we have to guard the occurrence of iProp by a later $\blacktriangleright$. Formally, we instantiate Iris with the following function:

$$F_{\text{INV}}(X) \triangleq \textsc{Auth}(\mathbb{N} \xrightarrow{\text{fin}} \textsc{Ag}(\blacktriangleright X))$$

- We are not interested in expressing exclusive ownership of invariants, like we did for heap locations. Instead, the entire point of invariants is *sharing*, so we need to make sure that everybody *agrees* on what the invariant with a given name is. To that end, we use the agreement camera AG instead of the exclusive camera EX.

In the definition of world satisfaction W and the invariant connective $\boxed{P}^{\iota}$, we use the elements of the camera $\textsc{Auth}(\mathbb{N} \xrightarrow{\text{fin}} \textsc{Ag}(\blacktriangleright \mathsf{iProp}))$ in the following ways:

- Ownership of the fragment $\circ\, [\iota \leftarrow \mathsf{ag}(\mathsf{next}(P))]$ states that P is registered as an invariant named ι, and
- ownership of the authoritative element $\bullet\, \mathsf{ag}(\mathsf{next}(I))$ states that I is the full map of all registered invariants.

The camera enjoys the following properties:

$$\big|\circ\,[\iota \leftarrow \mathsf{ag}(\mathsf{next}(P))]\big| = \circ\,[\iota \leftarrow \mathsf{ag}(\mathsf{next}(P))]$$
$$\mathcal{V}\big(\bullet\,\mathsf{ag}(\mathsf{next}(I)) \cdot \circ\,[\iota \leftarrow \mathsf{ag}(\mathsf{next}(P))]\big) \Rightarrow (\triangleright I(\iota) \Leftrightarrow \triangleright P)$$
$$\text{if } \iota \notin \mathrm{dom}(I) \quad \bullet\,\mathsf{ag}(\mathsf{next}(I)) \rightsquigarrow \bullet\,\mathsf{ag}(\mathsf{next}(I[\iota \leftarrow P])) \cdot \circ\,[\iota \leftarrow \mathsf{ag}(\mathsf{next}(P))])$$

These properties, in turn, give rise to the following rules:

$$\boxed{P}^{\iota} \vdash \Box\boxed{P}^{\iota} \qquad \text{(INVREG-PERSIST)}$$
$$\boxed{\bullet\,\mathsf{ag}(\mathsf{next}(I))}^{\gamma_{\text{INV}}} * \boxed{P}^{\iota} \vdash \triangleright I(\iota) \Leftrightarrow \triangleright P \qquad \text{(INVREG-AGREE)}$$
$$\iota \notin \mathrm{dom}(I) \wedge \boxed{\bullet\,\mathsf{ag}(\mathsf{next}(I))}^{\gamma_{\text{INV}}} \vdash \dot{\Rightarrow}\big(\boxed{\bullet\,\mathsf{ag}(\mathsf{next}(I[\iota \leftarrow P]))}^{\gamma_{\text{INV}}} * \boxed{P}^{\iota}\big) \qquad \text{(INVREG-ALLOC)}$$

The rule INVREG-PERSIST follows from the definition of the core $|-|$ of fragments. The rule INVREG-AGREE witnesses that the registry and the fragments *agree* on the proposition managed at a particular name. Note that we only get the bi-implication with a later $\triangleright$ because the occurrence of iProp in $\textsc{Auth}(\mathbb{N} \xrightarrow{\text{fin}} \textsc{Ag}(\blacktriangleright\mathsf{iProp}))$ is necessarily guarded with a later $\blacktriangleright$. This rule follows from the property of validity of composing authoritative and fragmentary elements. Finally, INVREG-ALLOC can be used to create a new invariant, provided the new name is not already used. This rule follows from the frame-preserving update.

Naively, we may think that world satisfaction should always require all invariants to hold. However, this does not work: after all, within a single atomic step of execution, threads are allowed to *temporarily break* invariants so long as they restore them by the end of the step. To support this, world satisfaction keeps invariants in one of two states: either they are *enabled* (currently enforced), or they are *disabled* (currently broken by some thread). The definition of the weakest precondition connective will then ensure that invariants are never disabled for more than an atomic step. That is, no invariant is left disabled between physical computation steps.

The protocol for opening (*i.e.,* disabling) and closing (*i.e.,* re-enabling) an invariant employs two *exclusive tokens*: an *enabled* token $\boxed{\{\iota\}}^{\gamma_{\text{EN}}}$, which witnesses that the invariant is currently enabled and gives the right to disable it; and dually, a *disabled* token $\boxed{\{\iota\}}^{\gamma_{\text{DIS}}}$. These tokens are controlled by the following two simple RAs:

$$\textsc{En} \triangleq \wp(\mathbb{N}) \mid \lightning \qquad\qquad \textsc{Dis} \triangleq \wp^{\text{fin}}(\mathbb{N}) \mid \lightning$$

The composition for both RAs is disjoint union. The *invalid* elements $\lightning$ account for composition of overlapping sets.

Let us take a look at the definition of world satisfaction W again to see how these tokens are being used. For each invariant ι we either have that it is enabled and maintained—in which case world satisfaction *owns* $\triangleright I(\iota)$—or the invariant is disabled. Additionally, in the reverse of what one might naively expect, world satisfaction owns the disabled token $\boxed{\{\iota\}}^{\gamma_{\text{DIS}}}$ when ι is enabled, and it owns the enabled token $\boxed{\{\iota\}}^{\gamma_{\text{EN}}}$ when ι is disabled. We will see shortly why this achieves the desired semantics for enabled/disabled tokens. It is important to note that $\boxed{P}^{\iota}$ just means that the registry maps ι to P; it does not say anything about whether ι is enabled or not.

Properties of world satisfaction. With this encoding, we can prove the following key properties modeling the allocation, opening, and closing of invariants:

WSAT-ALLOC
$$\frac{\mathcal{E} \text{ is infinite}}{W * \triangleright P \twoheadrightarrow \dot{\Rrightarrow}\left(W * \exists \iota \in \mathcal{E}.\, \boxed{P}^{\iota}\right)}$$

WSAT-OPENCLOSE
$$\boxed{P}^{\iota} \vdash W * \boxed{\{\iota\}}^{\gamma_{\text{EN}}} \Leftrightarrow W * \triangleright P * \boxed{\{\iota\}}^{\gamma_{\text{DIS}}}$$

Let us look at the proof of the direction $\boxed{P}^{\iota} \vdash W * \boxed{\{\iota\}}^{\gamma_{\text{EN}}} \Rightarrow W * \triangleright P * \boxed{\{\iota\}}^{\gamma_{\text{DIS}}}$ of WSAT-OPENCLOSE in more detail. We start by using INVREG-AGREE to learn that the authoritative registry I maintained by world satisfaction contains our invariant P at index ι. We thus obtain from the big separating conjunction that $\triangleright P * \boxed{\{\iota\}}^{\gamma_{\text{DIS}}} \vee \boxed{\{\iota\}}^{\gamma_{\text{EN}}}$. Since we moreover own the enabled token $\boxed{\{\iota\}}^{\gamma_{\text{EN}}}$, we can exclude the right disjunct and deduce that the invariant is currently enabled. So we take out the $\triangleright P$ and the disabled token, and instead put the enabled token into W, disabling the invariant. This concludes the proof.

The proof of WSAT-ALLOC is slightly more subtle. In particular, we have to be careful in picking the new invariant name such that: (a) it is in $\mathcal{E}$, (b) it is not used in I yet, and (c) we can create a disabled token for that name and put it into W alongside $\triangleright P$. Since disabled tokens are modeled by *finite* sets, only finitely many of them can ever be allocated, so it is always possible to pick an appropriate fresh name.

7.1.2 Invariant namespaces

We have defined the proposition $\boxed{P}^{\iota}$ expressing knowledge that P is maintained as an invariant with name ι. This way of naming invariants has two limitations:

1. Invariant names are unstructured. In a proof that uses n invariants, we will have n names. If we wish to open any combination of these invariants at the same time, we have to keep track of n^2 mutual inequalities to ensure that all names are different.
2. To make the situation worse, the concrete names of invariants are picked when invariants are allocated, so they cannot possibly be statically known (see the existential quantifier in WSAT-ALLOC).

To solve these issues, we organize invariants in *namespaces* $\mathcal{N} \in \mathit{List}(\mathbb{N})$. Namespaces can be thought of as fully qualified class names, and as such have a tree-like structure. For example, `org.iris` has sub-namespaces like `org.iris.lock` and `org.iris.barrier`.

The crux to solving (2) is that each namespace $\mathcal{N}$ *contains* infinitely many names, as given by its closure $\mathcal{N}^{\uparrow} \subseteq \mathbb{N}$:

$$\mathcal{N}^{\uparrow} \triangleq \left\{ f(\mathcal{N}') \;\middle|\; \mathcal{N}' \text{ is a sub-namespace of } \mathcal{N} \right\}$$

Here, f is an injection from $\mathit{List}(\mathbb{N})$ to $\mathbb{N}$, which exists because $\mathit{List}(\mathbb{N})$ is countably infinite. We now define:

$$\boxed{P}^{\mathcal{N}} \triangleq \exists \iota \in \mathcal{N}^{\uparrow}.\, \boxed{P}^{\iota}$$

Let us see how this definition solves the aforementioned issues:

1. Whenever namespaces $\mathcal{N}_1$ and $\mathcal{N}_2$ are not sub-namespaces of one another, we know that $\mathcal{N}_1^{\uparrow} \cap \mathcal{N}_2^{\uparrow} = \emptyset$, and as such, they can be opened at the same time. For example, the namespaces $\mathcal{N}$`.lock` and $\mathcal{N}$`.barrier` represent disjoint sets of invariant names no matter the choice of $\mathcal{N}$.

2. Since each namespace $\mathcal{N}$ represents an infinite set $\mathcal{N}^\uparrow$, once we allocate an invariant, we can use any $\iota \in \mathcal{N}^\uparrow$. This gives rise to the following rule:

$$\text{WSAT-ALLOC-NAMESP}\quad W * \triangleright P \twoheadrightarrow \dot{\Rightarrow}\left(W * \boxed{P}^{\mathcal{N}}\right)$$

In particular, the existential quantifier is gone (or rather, it is hidden). From now on, we will write $\mathcal{N}$ instead of $\mathcal{N}^\uparrow$ when there is no ambiguity—in particular, when $\mathcal{N}$ is used where a mask $\mathcal{E}$ is expected.

7.1.3 Cancellable invariants

Invariants as described above are persistent: Once allocated, they must be satisfied forever. On first sight, it may seem as if this would lead to problems when reasoning about code that performs memory deallocation. For example, a lock that can be deallocated must "tear down" its invariant when the lock is destroyed. However, Iris's ghost state is powerful enough to encode such *cancellable* invariants—*i.e.,* invariants that come with an additional mechanism to tear down the invariant and re-gain full ownership of the resources protected by the invariant. Indeed, cancellable invariants have already proven crucial in work on Iris that deals with languages supporting deallocation (Jung *et al.*, 2018; Kaiser *et al.*, 2017). Their encoding is formally described in our appendix (Iris Team, 2017).

7.2 View shifts and the fancy update modality

Before we define weakest preconditions and prove the rules for invariants, there is one other piece of the Iris program logic we need to cover: view shifts ($\Rrightarrow$). These serve three roles:

1. They permit frame-preserving updates, as realized by the rule GHOST-UPDATE:

$$\frac{a \rightsquigarrow B}{\boxed{a}^\gamma \Rrightarrow_{\mathcal{E}} \exists b \in B.\, \boxed{b}^\gamma}$$

2. They allow stripping away a later modality off a timeless proposition, as realized by the rule VS-TIMELESS:

$$\frac{\mathsf{timeless}(P)}{\triangleright P \Rrightarrow_{\mathcal{E}} P}$$

3. They allow accessing invariants, as realized by the following rule, which we have not shown before:

$$\frac{\triangleright P * Q_1 \Rrightarrow_{\mathcal{E}\setminus\mathcal{N}} \triangleright P * Q_2 \qquad \mathcal{N} \subseteq \mathcal{E}}{\boxed{P}^{\mathcal{N}} * Q_1 \Rrightarrow_{\mathcal{E}} \boxed{P}^{\mathcal{N}} * Q_2}$$

We have already seen two connectives that can be used for the first *or* second role of view shifts, respectively: the basic update modality $\dot{\Rightarrow}$, which can be used to perform frame-preserving updates (UPD-UPDATE), and the except-0 modality $\diamond$, which can be used to strip laters of timeless propositions ($\diamond$-TIMELESS).

$$\textsc{fup-mono}\ \frac{P \vdash Q}{{}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} P \vdash {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} Q} \qquad \textsc{fup-intro-mask}\ \frac{\mathcal{E}_2 \subseteq \mathcal{E}_1}{\mathsf{True} \vdash {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} {}^{\mathcal{E}_2}\!\Rrightarrow^{\mathcal{E}_1} \mathsf{True}} \qquad \textsc{fup-trans}\ {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} {}^{\mathcal{E}_2}\!\Rrightarrow^{\mathcal{E}_3} P \vdash {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_3} P$$

$$\textsc{fup-frame}\ Q * {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} P \vdash {}^{\mathcal{E}_1 \uplus \mathcal{E}_f}\!\Rrightarrow^{\mathcal{E}_2 \uplus \mathcal{E}_f} (Q * P) \qquad \textsc{fup-upd}\ \dot{\Rrightarrow} P \vdash \Rrightarrow_{\mathcal{E}} P \qquad \textsc{fup-timeless}\ \frac{\mathsf{timeless}(P)}{\triangleright P \vdash \Rrightarrow_{\mathcal{E}} P} \qquad \textsc{inv-persist}\ \boxed{P}^{\mathcal{N}} \vdash \square \boxed{P}^{\mathcal{N}}$$

$$\textsc{inv-alloc}\ \triangleright P \vdash \Rrightarrow_{\mathcal{E}} \boxed{P}^{\mathcal{N}} \qquad \textsc{inv-access}\ \frac{\mathcal{N} \subseteq \mathcal{E}}{\boxed{P}^{\mathcal{N}} \vdash {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E} \setminus \mathcal{N}} \left(\triangleright P * (\triangleright P \mathrel{-\!\!*} {}^{\mathcal{E} \setminus \mathcal{N}}\!\Rrightarrow^{\mathcal{E}} \mathsf{True})\right)}$$

Fig. 15. Rules for the fancy update modality and invariants.

We will now define a new modality that subsumes the basic update modality $\dot{\Rrightarrow}$ and the except-0 modality $\diamond$ and can also take care of the last role: accessing invariants. This modality, called the *fancy update modality* ${}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2}$, is defined as follows:

$$\begin{aligned} {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} P &\triangleq W * \lceil \overline{\mathcal{E}_1} \rceil^{\gamma_{\mathrm{EN}}} \mathrel{-\!\!*} \dot{\Rrightarrow} \diamond (W * \lceil \overline{\mathcal{E}_2} \rceil^{\gamma_{\mathrm{EN}}} * P) \\ \Rrightarrow_{\mathcal{E}} P &\triangleq {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}} P \end{aligned}$$

Before going into details about the definition or the proof rules (see Figure 15) of this modality, let us mention that in the same way that Hoare triples are defined in terms of weakest preconditions, view shifts are defined in terms of fancy updates:

$$\begin{aligned} P \ {}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} Q &\triangleq \square (P \mathrel{-\!\!*} {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} Q) \\ P \Rightarrow_{\mathcal{E}} Q &\triangleq P \ {}^{\mathcal{E}}\!\Rightarrow^{\mathcal{E}} Q \end{aligned}$$

Using the rules of the fancy update modality in Figure 15, it is easy to establish the rules for view shifts that we have seen in this paper.

The intuition behind ${}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} P$ is to express ownership of resources such that, if we further assume that the invariants in $\mathcal{E}_1$ are enabled, we can perform a *frame-preserving update* such that we end up owning P and the invariants in $\mathcal{E}_2$ are enabled. By looking at the definition of ${}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} P$, we can see how it supports all the fancy roles of view shifts:

1. At the heart of the fancy update modality is a basic update modality, which permits doing frame-preserving updates (see the rule FUP-UPD in Figure 15).
2. The modality is able to remove laters from timeless propositions by incorporating the "except 0" modality $\diamond$ (see §5.7 and FUP-TIMELESS).
3. Finally, the modality "threads through" world satisfaction, in the sense that a proof of ${}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} P$ can use W but also has to reestablish it. Furthermore, controlled by the two masks $\mathcal{E}_1$ and $\mathcal{E}_2$, the modality provides and consumes "enabled" tokens. The first mask controls which invariants are available to the modality, while the second mask controls which invariants remain available afterwards (see INV-ACCESS). It is also possible to allocate new invariants (INV-ALLOC). Notice that this is mostly a matter of convenience; treating the tokens as masks instead of the underlying ghost resources keeps them out of the way for the majority of the proof that does not care about them.

The rules FUP-MONO, FUP-TRANS, and FUP-FRAME correspond to the related rules of the basic update modality in Figure 11. We can also derive a rule corresponding to UPD-INTRO:

$$\text{FUP-INTRO}\quad P \vdash \Rrightarrow_{\mathcal{E}} P$$

This follows from FUP-INTRO-MASK, FUP-TRANS and FUP-FRAME (or, alternatively, from FUP-UPD).

It is important to notice that there is *no* rule like $P \vdash {}^{\mathcal{E}_1}\Rrightarrow^{\mathcal{E}_2} P$ that lets us introduce an arbitrary *mask-changing* fancy update. With a goal of the form ${}^{\mathcal{E}_1}\Rrightarrow^{\mathcal{E}_2}$, the fact that the masks are different actually presents an *obligation* that we have to discharge—typically by eliminating fancy update modalities until the masks *do* match. The reason this works is that elimination of a mask-changing fancy update changes only the *first* of the two masks in our goal:

$$\text{FUP-CHANGE-MASK}\quad \frac{R * P \vdash {}^{\mathcal{E}_2}\Rrightarrow^{\mathcal{E}_3} Q}{R * {}^{\mathcal{E}_1}\Rrightarrow^{\mathcal{E}_2} P \vdash {}^{\mathcal{E}_1}\Rrightarrow^{\mathcal{E}_3} Q}$$

As you can see, eliminating an update with masks $\mathcal{E}_1$ and $\mathcal{E}_2$ has the effect of changing the first mask in the conclusion from $\mathcal{E}_1$ to $\mathcal{E}_2$, while the second mask ($\mathcal{E}_3$) stays unaffected. FUP-CHANGE-MASK follows from FUP-FRAME, FUP-MONO and FUP-TRANS.

As a more concrete example for how to work with mask-changing updates, we will prove the rule INV-ACCESS in Figure 15. The rule may look fairly cryptic; we will see in the next section how it can be used to derive WP-INV. The proof of this rule relies on the following two mask-changing updates, both centering around WSAT-OPENCLOSE:

$$\text{INV-OPEN}\quad \frac{\iota \in \mathcal{E}}{\boxed{P}^{\iota} \;\; {}^{\mathcal{E}}\Rrightarrow^{\mathcal{E}\setminus\{\iota\}} \triangleright P * \boxed{\{\iota\}}^{\gamma_{\text{DIS}}}}$$

$$\text{INV-CLOSE}\quad \frac{\iota \in \mathcal{E}}{\boxed{P}^{\iota} * \triangleright P * \boxed{\{\iota\}}^{\gamma_{\text{DIS}}} \;\; {}^{\mathcal{E}\setminus\{\iota\}}\Rrightarrow^{\mathcal{E}} \mathsf{True}}$$

We discuss the proof of INV-OPEN in some more detail; the proof of INV-CLOSE proceeds in a very similar way.

- To prove INV-OPEN, we start by unfolding the definition of the fancy update modality. We can thus assume $\boxed{P}^{\iota}$, W, and the enabled token $\boxed{\mathcal{E}}^{\gamma_{\text{EN}}}$. From these assumptions we now have to show $\dot{\Rrightarrow} W * \boxed{\mathcal{E}\setminus\iota}^{\gamma_{\text{EN}}} * \triangleright P * \boxed{\{\iota\}}^{\gamma_{\text{DIS}}}$.
- We split the enabled token $\boxed{\mathcal{E}}^{\gamma_{\text{EN}}}$ into $\boxed{\{\iota\}}^{\gamma_{\text{EN}}}$ and $\boxed{\mathcal{E}\setminus\{\iota\}}^{\gamma_{\text{EN}}}$.
- We do not actually need to do any frame-preserving updates, so we just apply UPD-INTRO. Furthermore, we have the assumptions to use WSAT-OPENCLOSE, which gives us W, the disabled token $\boxed{\{\iota\}}^{\gamma_{\text{DIS}}}$, and $\triangleright P$. We can thus easily discharge our goal.

Now we can prove INV-ACCESS using just the rules INV-OPEN, INV-CLOSE, and some rules shown in Figure 15. Below we give detailed notes to accompany the derivation in Figure 16.

1. We begin by assuming $\boxed{P}^{\mathcal{N}}$, from which we have to show ${}^{\mathcal{E}}\Rrightarrow^{\mathcal{E}\setminus\mathcal{N}} \triangleright P * (\triangleright P \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\mathcal{N}}\Rrightarrow^{\mathcal{E}} \mathsf{True})$. By definition of $\boxed{P}^{\mathcal{N}}$ this means there exists $\iota \in \mathcal{N}$ s.t. $\boxed{P}^{\iota}$. So using INV-OPEN and $\boxed{P}^{\iota}$, we can add ${}^{\mathcal{E}}\Rrightarrow^{\mathcal{E}\setminus\{\iota\}} \triangleright P * \boxed{\{\iota\}}^{\gamma_{\text{DIS}}}$ to our assumptions. We can keep $\boxed{P}^{\iota}$ around because it is persistent.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\boxed{P}^{\mathcal{N}} \vdash {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}\setminus\mathcal{N}} \triangleright P * (\triangleright P \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\mathcal{N}}\!\Rrightarrow^{\mathcal{E}} \mathsf{True})}\,(1)}{\boxed{P}^{\iota} * \left({}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}\setminus\{\iota\}} \triangleright P * \boxed{\{\iota\}}^{\gamma_{\mathrm{DIS}}}\right) \vdash {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}\setminus\mathcal{N}} \triangleright P * (\triangleright P \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\mathcal{N}}\!\Rrightarrow^{\mathcal{E}} \mathsf{True})}\,(2)}{\boxed{P}^{\iota} * \triangleright P * \boxed{\{\iota\}}^{\gamma_{\mathrm{DIS}}} \vdash {}^{\mathcal{E}\setminus\{\iota\}}\!\Rrightarrow^{\mathcal{E}\setminus\mathcal{N}} \triangleright P * (\triangleright P \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\mathcal{N}}\!\Rrightarrow^{\mathcal{E}} \mathsf{True})}\,(3)}{\boxed{P}^{\iota} * \triangleright P * \boxed{\{\iota\}}^{\gamma_{\mathrm{DIS}}} * \left({}^{\mathcal{E}\setminus\{\iota\}}\!\Rrightarrow^{\mathcal{E}\setminus\mathcal{N}} {}^{\mathcal{E}\setminus\mathcal{N}}\!\Rrightarrow^{\mathcal{E}\setminus\{\iota\}} \mathsf{True}\right) \vdash {}^{\mathcal{E}\setminus\{\iota\}}\!\Rrightarrow^{\mathcal{E}\setminus\mathcal{N}} \triangleright P * (\triangleright P \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\mathcal{N}}\!\Rrightarrow^{\mathcal{E}} \mathsf{True})}\,(4)}{\boxed{P}^{\iota} * \triangleright P * \boxed{\{\iota\}}^{\gamma_{\mathrm{DIS}}} * {}^{\mathcal{E}\setminus\mathcal{N}}\!\Rrightarrow^{\mathcal{E}\setminus\{\iota\}} \mathsf{True} \vdash \triangleright P * (\triangleright P \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\mathcal{N}}\!\Rrightarrow^{\mathcal{E}} \mathsf{True})}\,(5)}{\boxed{P}^{\iota} * \triangleright P * \boxed{\{\iota\}}^{\gamma_{\mathrm{DIS}}} * {}^{\mathcal{E}\setminus\mathcal{N}}\!\Rrightarrow^{\mathcal{E}\setminus\{\iota\}} \mathsf{True} \vdash {}^{\mathcal{E}\setminus\{\iota\}}\!\Rrightarrow^{\mathcal{E}} \mathsf{True}}\,(6)}{\boxed{P}^{\iota} * \triangleright P * \boxed{\{\iota\}}^{\gamma_{\mathrm{DIS}}} * \mathsf{True} \vdash {}^{\mathcal{E}\setminus\{\iota\}}\!\Rrightarrow^{\mathcal{E}} \mathsf{True}}\,(7)}{{}^{\mathcal{E}\setminus\{\iota\}}\!\Rrightarrow^{\mathcal{E}} \mathsf{True} \vdash {}^{\mathcal{E}\setminus\{\iota\}}\!\Rrightarrow^{\mathcal{E}} \mathsf{True}}$$

Fig. 16. Derivation of INV-ACCESS.

2. Now we apply FUP-CHANGE-MASK, which means the update modality is removed from our new assumption and the goal changes to ${}^{\mathcal{E}\setminus\{\iota\}}\!\Rrightarrow^{\mathcal{E}\setminus\mathcal{N}} \ldots$. Observe how, by applying view shift INV-OPEN, which has two *different* masks, the first mask in our goal changes from $\mathcal{E}$ to $\mathcal{E} \setminus \{\iota\}$.
3. We do not have any more invariants to open, but our goal is still a mask-changing update. This is where we use FUP-INTRO-MASK. We proceed in much the same way as when we applied INV-OPEN in the previous step: First, we obtain a new assumption ${}^{\mathcal{E}\setminus\{\iota\}}\!\Rrightarrow^{\mathcal{E}\setminus\mathcal{N}} {}^{\mathcal{E}\setminus\mathcal{N}}\!\Rrightarrow^{\mathcal{E}\setminus\{\iota\}} \mathsf{True}$.
4. Next, we apply FUP-CHANGE-MASK, which eliminates the outer of the two update modalities from our new assumption, and turns our goal into ${}^{\mathcal{E}\setminus\mathcal{N}}\!\Rrightarrow^{\mathcal{E}\setminus\mathcal{N}} \ldots$. Now we can use FUP-INTRO to get rid of the fancy update in the goal, because the two masks are the same.
5. Since we have $\triangleright P$ as an assumption, we can now discharge the left-hand side of our goal. For the right-hand side, we immediately assume $\triangleright P$ again, and our goal becomes ${}^{\mathcal{E}\setminus\mathcal{N}}\!\Rrightarrow^{\mathcal{E}} \mathsf{True}$.
6. Using FUP-CHANGE-MASK with our assumption ${}^{\mathcal{E}\setminus\mathcal{N}}\!\Rrightarrow^{\mathcal{E}\setminus\{\iota\}} \mathsf{True}$ (which we obtained from the update we got via FUP-INTRO-MASK), the goal changes to ${}^{\mathcal{E}\setminus\{\iota\}}\!\Rrightarrow^{\mathcal{E}} \mathsf{True}$.
7. The goal now immediately follows from INV-CLOSE and our remaining assumptions.

7.3 Weakest preconditions

We will now define weakest preconditions that support not only the rules in Figure 13, but also the ones in Figure 17. We will also show how, from WP-ATOMIC and INV-ACCESS, we can derive the rule motivating this entire section, WP-INV.

Compared to the definition developed in §6, there are two key differences: First of all, we use the fancy update modality instead of the basic update modality. Secondly, we do not want to tie the definition of weakest preconditions to a particular language, and instead we operate generically over any notion of expressions and state, and any reduction relation.

The definitions in this section are thus parameterized by a language, which consists of:

- A set of expressions *Expr*,
- A set of values $Val \subseteq Expr$,

WP-VUP
$$\Rrightarrow_{\mathcal{E}} \mathsf{wp}_{\mathcal{E}}\, e\, \{v.\, \Rrightarrow_{\mathcal{E}} \Phi(v)\} \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}$$

WP-ATOMIC
$$\frac{\mathsf{atomic}(e)}{{}^{\mathcal{E}_1}\Rrightarrow^{\mathcal{E}_2} \mathsf{wp}_{\mathcal{E}_2}\, e\, \left\{v.\, {}^{\mathcal{E}_2}\Rrightarrow^{\mathcal{E}_1} \Phi(v)\right\} \vdash \mathsf{wp}_{\mathcal{E}_1}\, e\, \{\Phi\}}$$

Fig. 17. New rules for weakest precondition with invariants.

- A set of states *State*, and
- A per-thread step relation $(\rightarrow_{\mathrm{t}}) \subseteq (\mathit{Expr} \times \mathit{State}) \times (\mathit{Expr} \times \mathit{State} \times \mathit{List}(\mathit{Expr}))$.

These should satisfy: if $(e, \sigma) \rightarrow_{\mathrm{t}} (e', \sigma', \vec{e}_f)$ then $e \notin \mathit{Val}$.

As a consequence of parameterizing by a language, we can no longer assume that our physical state is a heap of values with disjoint union as composition. Therefore, instead of using the authoritative heap defined in §6.3, we further parameterize weakest preconditions by a predicate $S : \mathit{State} \rightarrow \mathit{iProp}$ called the *state interpretation*. In case $\mathit{State} = \mathit{Loc} \xrightarrow{\mathrm{fin}} \mathit{Val}$, we can recover the definition and rules from §6.3 by taking:

$$S(\sigma) \triangleq \boxed{\bullet \left(\sigma : \mathit{Loc} \xrightarrow{\mathrm{fin}} \mathrm{Ex}(\mathit{Val})\right)}^{\gamma_{\mathrm{HEAP}}}$$

More sophisticated forms of separation like fractional permissions (Boyland, 2003; Bornat *et al.*, 2005) can be obtained by using an appropriate camera and defining S accordingly.

Given a state interpretation $S : \mathit{State} \rightarrow \mathit{iProp}$, our definition of weakest precondition looks as follows (changes from §6.3.4 are colored red):

$$\begin{aligned}\mathsf{wp}^{S}_{\mathcal{E}}\, e\, \{\Phi\} \triangleq\ & (e \in \mathit{Val} \wedge \Rrightarrow_{\mathcal{E}} \Phi(e)) \\ & \vee \Big(e \notin \mathit{Val} \wedge \forall \sigma.\, S(\sigma) \mathrel{-\!\!*} {}^{\mathcal{E}}\Rrightarrow^{\emptyset} \left(\mathrm{red}(e, \sigma)\right. \\ & \quad \wedge \triangleright \forall e_2, \sigma_2, \vec{e}_f.\, \left((e, \sigma) \rightarrow_{\mathrm{t}} (e_2, \sigma_2, \vec{e}_f)\right) \mathrel{-\!\!*} {}^{\emptyset}\Rrightarrow^{\mathcal{E}} \\ & \quad \left.\left(S(\sigma_2) * \mathsf{wp}^{S}_{\mathcal{E}}\, e_2\, \{\Phi\} * \mathop{\Large *}_{e' \in \vec{e}_f} \mathsf{wp}^{S}_{\top}\, e'\, \{v.\mathsf{True}\}\right)\right)\Big)\end{aligned}$$

The state interpretation S is generally clear from the context, so we often leave it implicit and write just $\mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}$. The mask $\mathcal{E}$ of $\mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}$ is used for the "outside" of the fancy update modalities, providing them with access to these invariants. The "inner" masks are $\emptyset$, indicating that the reasoning about safety and progress can temporarily open/disable *all* invariants. The forked-off threads $\vec{e}_f$ have access to the full mask $\top$ as they will only start running in the *next* instruction, so they are not constrained by whatever invariants are available right now. Note that the definition requires all invariants in $\mathcal{E}$ to be enabled again after every physical step: This corresponds to the fact that an invariant can only be opened atomically.

In addition to the rules already presented in §6, this version of the weakest precondition connective lets us prove (among others) the new rules in Figure 17. The rule WP-VUP witnesses that the entire connective as well as its postcondition are living below the fancy update modality, so we can freely add/remove that modality.

Finally, we come to WP-ATOMIC to open an invariant around an atomic expression. The rule is similar to WP-VUP, with the key difference being that it can *change the mask*. On the left-hand side of the turnstile, we are allowed to first open some invariants, then reason about e, and then close invariants again. This is sound because e is atomic. The rule WP-ATOMIC is needed to derive WP-INV:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\triangleright P \vdash \mathsf{wp}_{\mathcal{E}\setminus\mathcal{N}}\, e\, \{v.\, \triangleright P * \Phi(v)\}}{\triangleright P * (\triangleright P \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\mathcal{N}}\!\Rightarrow^{\mathcal{E}} \mathsf{True}) \vdash \mathsf{wp}_{\mathcal{E}\setminus\mathcal{N}}\, e \left\{v.\, \triangleright P * \Phi(v) * (\triangleright P \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\mathcal{N}}\!\Rightarrow^{\mathcal{E}} \mathsf{True})\right\}}\,\text{WP-FRAME}}{\triangleright P * (\triangleright P \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\mathcal{N}}\!\Rightarrow^{\mathcal{E}} \mathsf{True}) \vdash \mathsf{wp}_{\mathcal{E}\setminus\mathcal{N}}\, e \left\{v.\, \Phi(v) * {}^{\mathcal{E}\setminus\mathcal{N}}\!\Rightarrow^{\mathcal{E}} \mathsf{True}\right\}}\,\text{WP-MONO}}{\triangleright P * (\triangleright P \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\mathcal{N}}\!\Rightarrow^{\mathcal{E}} \mathsf{True}) \vdash \mathsf{wp}_{\mathcal{E}\setminus\mathcal{N}}\, e \left\{v.\, {}^{\mathcal{E}\setminus\mathcal{N}}\!\Rightarrow^{\mathcal{E}} \Phi(v)\right\}}\,\text{FUP-FRAME}}{\boxed{P}^{\mathcal{N}} \vdash {}^{\mathcal{E}}\!\Rightarrow^{\mathcal{E}\setminus\mathcal{N}} \mathsf{wp}_{\mathcal{E}\setminus\mathcal{N}}\, e \left\{x.\, {}^{\mathcal{E}\setminus\mathcal{N}}\!\Rightarrow^{\mathcal{E}} \Phi(v)\right\}}\,\text{INV-ACCESS}}{\boxed{P}^{\mathcal{N}} \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}}\,\text{WP-ATOMIC}$$

7.4 Adequacy

To demonstrate that wp $e\,\{\phi\}$ actually makes the expected statements about program executions, we prove the following *adequacy theorem*.

Theorem 7 (Adequacy of weakest preconditions)**.** *Let ϕ be a first-order predicate. If $\mathit{True} \vdash \Rightarrow_{\top} \exists S : \mathit{State} \to \mathit{iProp}.\, S(\sigma) * \mathit{wp}^{S}_{\top}\, e\, \{\phi\}$, where S is the state interpretation predicate, and $(e, \sigma) \to^{*}_{\mathsf{tp}} (e'_1 \ldots e'_n, \sigma')$, then:*

1. *For any e'_i we have that either e'_i is a value or* $\mathrm{red}(e'_i, \sigma')$*;*
2. *If e'_1 (the main thread) is a value v, then $\phi(v)$.*

The proof of this theorem is similar to the proof of Theorem 6. A notable difference is that now, the user of the theorem has to pick the state interpretation predicate S, which is existentially quantified below an update modality in the theorem. The reason for that is that one typically has to allocate some ghost variables before the predicate can be defined. For example, one can only define the predicate below—the one we use for our simple ML-like language—after a ghost variable with name γ_{HEAP} has been allocated.

$$S(\sigma) \triangleq \boxed{\bullet \left(\sigma : \mathit{Loc} \xrightarrow{\text{fin}} \text{EX}(\mathit{Val})\right)}^{\gamma_{\text{HEAP}}}$$

8 Discussion

8.1 Past development of Iris

In this section we describe some notable differences between the presentation of Iris in the present paper and the presentation in earlier papers (Jung *et al.*, 2015; Jung *et al.*, 2016; Krebbers *et al.*, 2017a). This section does *not* give a comprehensive list of differences between the Iris papers, just some that we think are worth discussing.

OFEs versus COFEs. A key difference between the Iris model as described in this paper and in the Iris 2.0 paper (Jung *et al.*, 2016) is the move from COFEs to OFEs. Previously, we worked exclusively with COFEs, *i.e.,* all OFEs that we used were required to satisfy the completeness property (OFE-COMPL). However, more recently, we noticed that the completeness property is only needed in two places:

- America and Rutten's theorem (Theorem 2), which we use in the model of Iris (§4.6) to construct a solution to the recursive domain equation IRIS.

- Banach's fixed-point theorem (Theorem 4), which we use to model the guarded fixed-point operator $\mu x.\, P$ in the logic (§5.6).

To use the first theorem, the only OFE that needs to be complete is $\mathit{UPred}(M)$, and to use the second theorem, the only OFE that needs to be complete is $T_1 \to \ldots \to T_n \to \mathit{UPred}(M)$. These OFEs are complete regardless of whether the camera M or the OFEs T_i are complete. More precisely, $\mathit{UPred}(M)$ is complete even if M is not complete, and $T \xrightarrow{\text{ne}} U$ is complete even if T is not (because the limit is just taken pointwise in U).

A consequence of the move from COFEs to OFEs is that we no longer require cameras to be complete, which in turn allows us to use a much simpler construction for the higher-order agreement camera AG than we used previously.[16] This new definition of cameras furthermore enjoys a useful property that the old definition did not. It is now the case that if T is discrete (*i.e.,* the step-indexed equivalence is just equality: $\forall x, y, n.\, x \overset{n}{=} y \iff x = y$), then so is $\mathrm{AG}(T)$. As a consequence of this, ownership of ghost elements like $\mathsf{ag}(0)$ (in $\mathrm{AG}(\mathbb{Z})$) is timeless: a property we need in many of our proofs. The old higher-order agreement construction did not have this property, so we actually had two agreement constructions: AG_0 for timeless, first-order ghost state and the complicated version of AG for higher-order ghost state. The new AG presented in this paper unifies both constructions.

View shifts versus fancy updates. Iris 1.0 and 2.0 (Jung *et al.*, 2015; Jung *et al.*, 2016) use the (binary) view shift connective $P \;{}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} Q$ as the main vehicle for performing ghost updates, whereas Iris 3.0 (Krebbers *et al.*, 2017a) uses the (unary) fancy update modality ${}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} P$. It is worth noting that this is just a matter of presentation and does not affect the expressive power of the logic. As demonstrated in §7.2, view shifts $P \;{}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} Q$ can be encoded in terms of the fancy update modality as $P \mathrel{-\!\!*} {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} Q$. Perhaps more surprisingly, the fancy update modality can also be encoded in terms of view shifts:

$$ {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} P \triangleq \exists R.\, R * (R \;{}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} P) $$

However, ignoring this choice of presentation, there are actually some small semantic differences between view shifts in Iris 1.0/2.0 and 3.*x*. Firstly, in Iris 1.0/2.0, the rule FUP-TRANS had a side condition restricting the masks it could be instantiated with, whereas now it does not. Secondly, in Iris 1.0/2.0, instead of FUP-INTRO-MASK, only mask-invariant view shifts could be introduced (*i.e.,* there was only FUP-INTRO: $P \vdash \Rrightarrow_{\mathcal{E}} P$). The reason we can now support FUP-INTRO-MASK is that masks are actually just sugar for owning or providing particular *resources* (namely, the enabled tokens). This is in contrast to previous versions of Iris, where masks were entirely separate from resources and treated in a rather ad-hoc manner.

Our more principled treatment of masks significantly simplifies building abstractions involving invariants. In particular, recall the proof of INV-ACCESS in §7.2: The invariants in $\mathcal{N} \setminus \{\iota\}$ do, in fact, remain enabled even though they disappear from the mask. This is possible because FUP-INTRO-MASK lets us remove invariants from the mask and add them back later *without* actually opening them—effectively "hiding" enabled invariants. In Iris

[16] The conference paper (Jung *et al.*, 2016) did not even have the space for the full previous definition, which was only given in the appendix.

1.0/2.0, it would not have been possible to hide invariants like this. In fact, it was not possible to prove a concise abstract rule like INV-ACCESS for invariants with namespaces; we had to resort to a much more complicated lemma.

It is worth noting that as a consequence of defining weakest preconditions inside the base logic instead of directly in the model, we not only obtained a conceptual benefit, but also a very practical one: enabling projects that build on top of Iris to define their own variants of weakest preconditions. For example, Timany *et al.* (2018) use a version of weakest preconditions in big-step instead of small-step style, the Iris Coq development (Iris Team, 2017) also includes a version of weakest preconditions for total program correctness, Timany & Birkedal (2018) use a version of weakest preconditions for a language with continuations, and Tassarotti *et al.* (2018) use a version of weakest preconditions for randomized algorithms.

Differences from Iris 3.0. The differences between Iris 3.0 (Krebbers *et al.*, 2017a) and Iris 3.1 (this paper) are minor: The "always modality" $\Box$ has been renamed as the "persistence modality" in the interest of adopting a more consistent nomenclature. Moreover, we managed to derive two previously primitive proof rules about the persistence modality, namely $\mathsf{True} \vdash \Box\mathsf{True}$ and $\Box(P \wedge Q) \vdash \Box(P * Q)$, from the remaining rules. Finally, we slightly changed the definition of *UPred*, making it a quotient over monotone predicates instead of the previous, more ad-hoc (but isomorphic) definition.

8.2 Naive impredicative invariants paradox

In §3.3, we have seen that higher-order ghost state without the later modality (which is modeled using step-indexing) leads to a paradox. One may argue that the burden of having the later modality in the logic is too high and it is not worth having higher-order ghost state at that cost. However, we will see in this section that the later modality plays an *essential* role even without higher-order ghost state: It is needed for impredicative invariants. Indeed, we show here that omitting the later in the rule to access invariants (INV-ACCESS) leads to a contradiction, even in the absence of general higher-order ghost state.

This answers an open question: The later modality has been present in all concurrent separation logics with impredicative invariants (Svendsen & Birkedal, 2014; Jung *et al.*, 2015; Jung *et al.*, 2016), but it was unclear whether this was a mere artifact of the model, or whether it is in some sense required. We show that it is indeed required in some sense.

More precisely, we prove the following theorem, without using higher-order ghost state:

Theorem 8. *Assume we add the following proof rule to Iris:*

$$\frac{\iota \in \mathcal{E}}{\boxed{P}^{\iota} \vdash {}^{\mathcal{E}}\!\Rightarrow^{\mathcal{E}\setminus\{\iota\}} P * (P \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\{\iota\}}\!\Rightarrow^{\mathcal{E}} \mathsf{True})}$$

Then, if we pick an appropriate RA, $\mathsf{True} \vdash \Rightarrow_{\top} \mathsf{False}$.

Notice that the above rule is the same as INV-ACCESS in Figure 15, *except that it does not add a* $\triangleright$ *in front of* P.

Of course, this does not prove that we absolutely must have a $\triangleright$ modality, but it *does* show that the stronger rule one would prefer to have for invariants is unsound. The later

modality (and step-indexing) is but one way to navigate around this unsoundness. However, we are not aware of another technique that would yield a logic with comparably powerful impredicative invariants.

The proof of this theorem does not use the fact that fancy updates are defined in any particular way, but just uses the proof rules for the fancy update modality. In fact, the paradox can be generalized so that it applies to all versions of Iris (Jung *et al.*, 2015; Jung *et al.*, 2016; Krebbers *et al.*, 2017a), as is shown by the following theorem:

Theorem 9. *Assume a higher-order separation logic with* $\Box$ *and an update modality with a binary mask* $\Rightarrow_{\{0,1\}}$ *(think: empty mask and full mask) satisfying strong monad rules with respect to separating conjunction and such that:*

$$\textsc{weaken-mask}\quad \Rightarrow_0 P \vdash \Rightarrow_1 P$$

Assume a type $\mathcal{I}$ *and a proposition* $\boxed{\cdot}^{\cdot} : \mathcal{I} \to \mathit{iProp} \to \mathit{iProp}$ *satisfying:*

$$\textsc{inv-alloc}\quad P \vdash \Rightarrow_1 \exists \iota.\, \boxed{P}^{\iota} \qquad \textsc{inv-persist}\quad \boxed{P}^{\iota} \vdash \Box \boxed{P}^{\iota} \qquad \textsc{inv-open-nolater}\quad \frac{P * Q \vdash \Rightarrow_0 (P * R)}{\boxed{P}^{\iota} * Q \vdash \Rightarrow_1 R}$$

Finally, assume the existence of a type $\mathcal{G}$ *and two tokens* $\boxed{\mathrm{S}}^{\cdot} : \mathcal{G} \to \mathit{iProp}$ *and* $\boxed{\mathrm{F}}^{\cdot} : \mathcal{G} \to \mathit{iProp}$ *parameterized by* $\mathcal{G}$ *and satisfying the following properties:*

$$\textsc{start-alloc}\quad \vdash \Rightarrow_0 \exists \gamma.\, \boxed{\mathrm{S}}^{\gamma} \qquad \textsc{start-finish}\quad \boxed{\mathrm{S}}^{\gamma} \vdash \Rightarrow_0 \boxed{\mathrm{F}}^{\gamma} \qquad \textsc{start-not-finished}\quad \boxed{\mathrm{S}}^{\gamma} * \boxed{\mathrm{F}}^{\gamma} \vdash \mathit{False} \qquad \textsc{finished-dup}\quad \boxed{\mathrm{F}}^{\gamma} \vdash \boxed{\mathrm{F}}^{\gamma} * \boxed{\mathrm{F}}^{\gamma}$$

Then $\mathit{True} \vdash \Rightarrow_1 \mathit{False}$.

In other words, the theorem requires three ingredients to be present in the logic in order to derive a contradiction:

- An update modality that satisfies the laws of Iris's basic update modality (Figure 11). The modality needs a mask for the same reason that Iris's fancy update modality has a mask: to prevent opening the same invariant twice.
- Invariants that can be opened around the update modality *without* a later.
- A two-state protocol whose only transition is from the first to the last state. This is what $\boxed{\mathrm{S}}^{\cdot}$ and $\boxed{\mathrm{F}}^{\cdot}$ encode. The proof does not actually depend on how that protocol is made available to the logic. For example, to apply this proof to iCAP (Svendsen & Birkedal, 2014), one could use iCAP's built-in support for state-transition systems to achieve the same result. However, for the purpose of this theorem, we had to pick *some* way of expressing protocols. We picked the token-based approach common in Iris.

All versions of Iris easily satisfy the first and third requirement, by using fancy updates (Iris 3) or view shifts (Iris 1 and 2) for the update modality $\Rightarrow_{\{0,1\}}$, and by constructing an appropriate RA (Iris 2 and 3) or PCM (Iris 1) for the two-state protocol. Of course, INV-OPEN-NOLATER is the one assumption of the theorem that no version of Iris satisfies, which is the entire point.

The proof works by constructing a proposition that is equivalent (in some rather loose sense) to the stored propositions we used for the paradox in §3.3, and then deriving a

contradiction in much the same way. The full details of this construction are spelled out in the appendix (Iris Team, 2017).

8.3 Formalization in Coq

All results in this paper, including adequacy and soundness of Iris as well as the derived constructions like weakest preconditions and invariants, have been formalized using Coq. Additionally, we have formalized a large body of derived reasoning principles for concurrency in Iris. The Iris Coq formalization has been used by a variety of projects, *e.g.,* (Kaiser *et al.*, 2017; Swasey *et al.*, 2017; Jung *et al.*, 2018; Timany *et al.*, 2018; Timany & Birkedal, 2018; Tassarotti & Harper, 2018; Tassarotti *et al.*, 2017; Frumin *et al.*, 2018).

The Coq formalization of Iris is a so-called *shallow embedding* (Wildmoser & Nipkow, 2004). That is, the logical connectives are defined directly as functions on the semantic model of propositions *iProp*. This approach differs from a *deep embedding*—of the sort we have used in this paper—where one first defines an explicit syntax of the logic and then interprets that syntax into the semantic model. When working in Coq, avoiding the indirection of first defining a syntax has a couple of technical advantages:

- It allows us to use Coq functions to represent binding in the object logic (*e.g.,* in the logical quantifiers and postconditions of weakest preconditions) instead of having to use an explicit encoding of binders, like De Bruijn indices.
- It avoids the need to extend the signature of the object logic with types like lists, natural numbers, finite sets, finite maps, *etc.* A shallow embedding allows one to just reuse the data types of the meta-logic, *i.e.,* Coq.

Since Iris uses a step-indexed model instead of a set-theoretic model, the shallow embedding does not use plain Coq types and plain Coq functions, but rather OFEs and non-expansive functions. In order to make a shallowly embedded step-indexed logic usable in practice, it is crucial to have Coq automatically infer that given Coq types are OFEs, and that given Coq functions are non-expansive. For this, we use the approach that we outlined in the Iris 2.0 paper (Jung *et al.*, 2016): Coq's canonical structures mechanism (Garillot *et al.*, 2009) is used to automate dealing with OFEs, and Coq's setoid mechanism (Sozeau, 2009) is used to automate dealing with non-expansive functions.

All proofs of the derived constructions (in particular, everything related to invariants and weakest preconditions) are carried out by solely using the rules of the Iris base logic, *i.e.,* without making use of the interpretation of Iris in the model. To enforce that this is indeed the case, we have made sure that Coq does not unfold the logical connectives into their definitions in the model (*i.e.,* we made sure that the model is *opaque*).

In order to formalize the proofs in the Iris base logic, we have used the Iris Proof Mode (IPM) (Krebbers *et al.*, 2017b), which provides tactic support for reasoning with interesting combinations of separation-logic propositions and our modalities $\triangleright$, $\Rightarrow$, and $\Box$ (for example, as they arise in the definition of weakest preconditions and fancy updates).

9 Related work

In §9.1 we will discuss work that is related to the pillars of Iris 1.0: user-defined ghost state and invariants. Sections §9.2 and §9.3 cover work that is related to the new features of Iris

2.0: higher-order ghost state and the generalization of PCMs to RAs/cameras. In §9.4, we will compare with work that has similar aims to our Iris 3.*x* base logic. Finally, in §9.5, we discuss the point of why Iris is an affine/intuitionistic (as opposed to a linear/classical) separation logic.

9.1 User-defined ghost state and invariants

Iris's main vehicle to provide user-defined ghost state is the parameterization of the logic by a user-chosen partial commutative monoid (PCM) (which we generalized to a user-chosen camera in Iris 2.0). The use of PCMs for this purpose is not entirely surprising, after all: They have been used since the earliest models of separation logic. As such, already before Iris 1.0, a number of different models and logics started to employ PCMs as a way of characterizing more fine-grained notions of interference (Krishnaswami *et al.*, 2012; Dinsdale-Young *et al.*, 2013; Ley-Wild & Nanevski, 2013). The Views framework (Dinsdale-Young *et al.*, 2013), in particular, enables the user to tie ghost resources—which are represented by a user-defined PCM—to physical resources.

However, the Views framework is limited in that it effectively requires the user to bake in a fixed invariant that ties ghost resources to physical ones, with no logical support for layering further invariants on top of those ghost resources. Iris, in contrast, provides built-in logical support for user-defined invariants, which allows one to relate (ghost and physical) resources to each other. Relatedly, in the Verified Software Toolchain (VST) (Appel, 2014) one can pick a fixed PCM (in fact, a "separation algebra") for ghost state, from which a higher-order separation logic is constructed. However, primitive rules for the chosen ghost state have to be proven in the model of the logic, whereas those can be derived from a few fundamental proof rules in Iris.

The idea of relating (ghost and physical) resources through invariants was already present in works that appeared before Iris 1.0—for example, in Pilkiewicz and Pottier's work on "monotonic state" (Pilkiewicz & Pottier, 2011), Jensen and Birkedal's work on "fictional separation logic" (Jensen & Birkedal, 2012), and Krishnaswami *et al.*'s work on "superficially substructural types" (Krishnaswami *et al.*, 2012). All of these earlier works were restricted to the sequential setting, however.

In the concurrent setting, work prior to Iris 1.0, such as CaReSL (Turon *et al.*, 2013), iCAP (Svendsen & Birkedal, 2014), and TaDA (da Rocha Pinto *et al.*, 2014), has typically used a particular PCM construction to fix its ghost state. Iris's combination of user-defined ghost state and invariants can be used to encode the patterns of reasoning found in these logics, but with a simpler set of primitive mechanisms and proof rules.

Around the time Iris 1.0 was developed, several other logics integrated support for user-defined ghost state. Notably, GPS (Turon *et al.*, 2014) also used PCMs and FCSL used an abstraction called "concurroids" (Nanevski *et al.*, 2014).

An interesting difference in expressiveness between Iris and FCSL is the support for "hiding" of invariants. FCSL supports a certain kind of hiding, namely the ability to transfer some local state into an invariant (actually a "concurroid"), which is enforced during the execution of a single expression e, but after which the state governed by the invariant is returned to local control. Iris can support such hiding as well, via cancellable invariants (§7.1.3). Additionally, Iris allows a different kind of hiding, namely the ability to

hide invariants used by (nested) Hoare-triple specifications. For example, the higher-order function `mk_oneshot` from §2 returns two functions (`tryset` and `check`), whose Hoare-triple specification is only correct under the invariant I (which was established during execution of `mk_oneshot`). Since invariants in Iris are persistent propositions, I can be hidden, *i.e.,* it need not infect the specification of `mk_oneshot`, `tryset`, or `check`. To our knowledge, FCSL does not support hiding of this form.

Like Iris, Cohen *et al.* (2009) strive to provide a minimal basis for concurrent reasoning, but theirs is based on a ghost heap and two-state invariants, whereas ours is based on arbitrary ghost PCMs and one-state invariants. The two approaches are optimizing for different goals. Our logic is substructural and supports more expressive—*e.g.,* higher-order—specifications. Theirs is not, making it better suited to use with automated verification tools like SMT solvers.

9.2 Higher-order ghost state

As we have just seen, some logics prior to Iris 2.0, such as GPS (Turon *et al.*, 2014) and FCSL (Nanevski *et al.*, 2014), supported user-defined ghost state like Iris 1.0. However, these logics are all restricted to *first-order* ghost state, since the structure of the ghost state is defined and fixed before the logic is instantiated.

Other prior work employs a model that can be considered to involve *second-order* ghost state, *e.g.,* to justify their use of dynamically allocated locks (Gotsman *et al.*, 2007; Hobor *et al.*, 2008; Buisse *et al.*, 2011), regions (Svendsen & Birkedal, 2014), or invariants (Jung *et al.*, 2015). However, this ghost state has a fixed structure, rather than being an instance of a generic algebraic structure like cameras.

In some of the logics prior to Iris 2.0, even though the structure of ghost state is fixed, it can sometimes be adapted cleverly to support a range of proof patterns. For example, Dodds *et al.* (2016) have shown that the built-in "protocol" mechanism of the iCAP logic (Svendsen & Birkedal, 2014) can be repurposed to encode *named propositions* (§3.3), a functionality for which it was not originally intended. However, the encoding is somewhat artificial, must be verified by direct appeal to the model of iCAP, and does not scale to support general higher-order ghost state as introduced in Iris 2.0 (Jung *et al.*, 2016).

The Verified Software Toolchain (VST) (Appel, 2014) is another logic that is generic in the kind of resources. It applies "indirection theory" (Hobor *et al.*, 2010) to solve recursive domain equations that are very much like the equation IRIS in §4. Although VST has been demonstrated to support particular forms of second-order ghost state, it is not yet clear how precisely indirection theory compares with the categorical COFE-based approach. We believe that VST can potentially be generalized to support more general higher-order ghost state in the manner of Iris 2.0. Notably, VST has a notion of PCM-like structures that are compatible with *aging* (step-indexing) and seem to loosely correspond to cameras.

9.3 Generalizations of PCMs

In prior work, there have been several presentations of generalizations of PCMs that have "multiple units" or include a notion of a "duplicable core".

Dockins *et al.* (2009) introduced "multi-unit separation algebras", which, unlike PCMs, only demand the existence of a possibly different unit u_a with $u_a \cdot a = a$ for any element a. A notable difference between multi-unit separation algebras and RAs is that we present the monoidal operation $\cdot$ and core $|-|$ as functions, whereas they represent these operations as relations. Treating these operations as functions, as we do, allows one to reason equationally, which makes it more convenient to carry out proofs.

The terminology of a *(duplicable) core* has been adapted from Pottier, who introduced it in the context of "monotonic separation algebras" (Pottier, 2013). However, the axioms of Pottier's cores, as well as those of related notions in other work (Turon *et al.*, 2014; Appel, 2014), are somewhat different from the axioms of our notion of RAs. Some common properties appear in most of these works (either as axioms or as admissible rules): The core must produce a unit (RA-CORE-ID), be idempotent (RA-CORE-IDEM), and be a homomorphism, *i.e.,* be compatible with composition: $|a \cdot b| = |a| \cdot |b|$. The last property is stronger than our monotonicity axiom (RA-CORE-MONO), and as such, the axioms of RAs are weaker than those demanded by prior work. In fact, RAs are *strictly* weaker: One of our most important RA constructions, the *state-transition system* (STS),[17] has a core that is *not* a homomorphism. This shows that demanding the core to be a homomorphism, as prior work has done, rules out useful instances.

Another difference is the fact that our core may be partial, whereas in prior work it was always a total function. As discussed in §3.1, partial cores make it easier to compose RAs out of simpler constructions like sums.

Finally, Bizjak and Birkedal (2017) have shown that our notion of the core is precisely what is needed to get a well-behaved notion of persistent propositions.

9.4 Minimal base logic

The most notable attempt at mitigating the difficulty of the model construction of a state-of-the-art separation logic is Svendsen and Birkedal's work on the iCAP logic (2014). They defined the model of the iCAP logic in the internal logic of the topos of trees, which includes a *later* connective to reason about step-indexing in an abstract way, *i.e.,* without step-indexes appearing explicitly in the interpretations of the connectives of the logic in the model. However, their model of Hoare triples still involves explicit resource management, which ours does not.

On the other end of the spectrum, there has been some work on encoding binary logical relations in concurrent separation logics (Dreyer *et al.*, 2010; Turon *et al.*, 2013; Krogh-Jespersen *et al.*, 2017; Krebbers *et al.*, 2017b). These encodings of binary logical relations are relying on an ambient separation logic that already includes a plethora of high-level concepts, such as weakest preconditions and view shifts. Our goal, in contrast, is precisely to *define* these concepts in simpler terms.

FCSL (Sergey *et al.*, 2015) takes an opposite approach to our base logic. To ease reasoning about programs in a proof assistant, they avoid reasoning in separation logic as much as possible, and reason mostly in the ambient logic (in their case, the Coq logic). This requires

[17] The RA of STSs, as well as the example demonstrating that the core is not a homomorphism, is described in our technical appendix (Iris Team, 2017).

the model to stay as simple as possible; in particular, FCSL does not make use of step-indexing. As a consequence, they do not support impredicative invariants, which we believe are an important feature of Iris. For example, they are needed to model impredicative type systems (Krebbers *et al.*, 2017b) or to model a reentrant event loop library (Svendsen & Birkedal, 2014). Furthermore, as we have shown in recent work (Krebbers *et al.*, 2017b), one *can* actually reason conveniently in a separation logic in Coq, so the additional complexity of our model is hardly visible to users of our logic.

The Verified Software Toolchain (VST) (Appel, 2014) is a framework that provides machinery for constructing sophisticated higher-order separation logics with support for impredicative invariants in Coq. However, VST is not a logic and, as such, does not abstract over step-indices and resources. Defining a program logic in VST thus still requires significant manual management of such details, which are abstracted away when defining a program logic in the Iris base logic. Logics built using the VST typically fix a particular ghost state for their purpose and provide primitive rules for this particular ghost state, whereas Iris is designed for such proof rules to be derived *inside* a single logic from a few fundamental proof rules. As a result, Iris proofs using different ghost state can be safely composed (§3.2), whereas proofs carried out in different VST logics do not necessarily interoperate. Furthermore, VST has so far only been demonstrated in the context of sequential reasoning and coarse-grained (lock-based) concurrency (Beringer *et al.*, 2014), whereas the focus of Iris is on fine-grained concurrency.

9.5 Affine vs. linear separation logic

As we pointed out in §2, Iris is an *affine* separation logic in the sense that we can always *throw away* resources—*i.e.,* all resources may be used at most once. Formally speaking, this means that for every proposition P and Q we have:[18]

$$P * Q \vdash P \qquad \text{(SEP-AFFINE)}$$

Separation logics that enjoy the above rule are often called "intuitionistic", because they do not admit the rule of excluded middle ($\mathsf{True} \vdash P \lor \neg P$): From the rule of excluded middle and SEP-AFFINE one can show that separating conjunction is degenerate, *i.e.,* $P \land Q \Leftrightarrow P * Q$ for any P and Q. Conversely, separation logics that do not enjoy the rule SEP-AFFINE are often called "classical" or "linear" because resources should be used exactly once.

The reader may of course wonder why Iris is affine/intuitionistic instead of linear/classical. There are various reasons for this:

- Logics that incorporate the later modality do not admit the law of excluded middle despite their linearity: From the rule of excluded middle and LÖB-induction one can prove that $\triangleright$ is degenerate, *i.e.,* $\triangleright P \Leftrightarrow \mathsf{True}$ for any P.
- The traditional advantage of linear separation logic is the more precise tracking of resources: Not being able to throw resources away enables one to reason about the absence of resource leaks. For example, in traditional classical separation logic it is guaranteed that allocated memory has been freed since one cannot throw away

[18] This follows from the fact that True is a unit of the separating conjunction.

points-to predicates $\ell \mapsto v$. However, to our knowledge, this property has never been proven for any existing logic for *fine-grained* concurrency (Nanevski *et al.*, 2014; Svendsen & Birkedal, 2014; da Rocha Pinto *et al.*, 2014). Besides, for logics with impredicative invariants (Hobor *et al.*, 2008; Svendsen & Birkedal, 2014), despite linearity, it is possible to store an invariant "inside itself", making it in turn possible to leak invariants and all the resources they contain. Hence, in said logics, one does not formally gain anything from linearity in terms of tracking resource leaks.

Nevertheless, linearity can be useful even when reasoning in impredicative concurrent separation logics, *e.g.,* to establish refinement properties (Tassarotti *et al.*, 2017) or to manage obligations using resources (Bizjak *et al.*, 2018).

10 Ongoing and future development of Iris

In this paper, we have tried to paint a comprehensive picture of present-day Iris from the ground up, but this is certainly not the last word on the subject. The Iris framework is continuously under development, and although we do not expect major changes to the foundations of Iris as we have presented them, we have been actively exploring several extensions that would broaden the generality and expressive power of Iris even further.

First of all, we are exploring ways to incorporate linear resources (§9.5) into Iris. We would like to find a suitable generalization of Iris such that the user of Iris has control over which resources are affine or linear, along the lines of recent work by Cao *et al.* (2017). Furthermore, we would like to determine which restrictions are needed on impredicative invariants so that Iris can be used to reason about the absence of resource leaks.

Beyond linearity, we have experimented with adding a new "plainness modality" for assertions that do not depend on *any* resources, even duplicable ones. Such "plain" assertions enjoy stronger proof rules in conjunction with the update modalities (Timany *et al.*, 2018). We have worked on generalizing our Coq infrastructure, the interactive proof mode (Krebbers *et al.*, 2017b), to handle a larger variety of logics (Krebbers *et al.*, 2018), which led to a more modular definition of the Iris model. Finally, we have extended weakest preconditions in two ways: to support reasoning about programs that may get stuck, which is useful in applications that interact with untrusted code (Swasey *et al.*, 2017), and to support reasoning about total program correctness.

Acknowledgments

This research was supported in part by a European Research Council (ERC) Consolidator Grant for the project "RustBelt", funded under the European Union's Horizon 2020 Framework Programme (grant agreement no. 683289); and by the ModuRes Sapere Aude Advanced Grant from The Danish Council for Independent Research for the Natural Sciences (FNU).

We thank Philippa Gardner for helpful feedback, and Rose Hoberman for editing advice. We also give many thanks to all those who have contributed to the development, implementation, and applications of the Iris framework.

References

America, Pierre, & Rutten, Jan. (1989). Solving reflexive domain equations in a category of complete metric spaces. *Journal of Computer and System Sciences*, **39**(3), 343–375.

Appel, Andrew W. (2001). Foundational proof-carrying code. *Pages 247–256 of: LICS*.

Appel, Andrew W. (ed). (2014). *Program Logics for Certified Compilers*. Cambridge University Press.

Appel, Andrew W., & McAllester, David. (2001). An indexed model of recursive types for foundational proof-carrying code. *TOPLAS*, **23**(5), 657–683.

Appel, Andrew W., Melliès, Paul-André, Richards, Christopher, & Vouillon, Jérôme. (2007). A very modal model of a modern, major, general type system. *Pages 109–122 of: POPL*.

Ashcroft, Edward A. (1975). Proving assertions about parallel programs. *Journal of Computer and System Sciences*, **10**(1), 110–135.

Beringer, Lennart, Stewart, Gordon, Dockins, Robert, & Appel, Andrew W. (2014). Verified compilation for shared-memory C. *Pages 107–127 of: ESOP*. LNCS, vol. 8410.

Birkedal, Lars, Støvring, Kristian, & Thamsborg, Jacob. (2010). The category-theoretic solution of recursive metric-space equations. *TCS*, **411**(47), 4102–4122.

Birkedal, Lars, Møgelberg, Rasmus Ejlers, Schwinghammer, Jan, & Støvring, Kristian. (2011). First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *Pages 55–64 of: LICS*.

Bizjak, Aleš, & Birkedal, Lars. (2017). On models of higher-order separation logic. *MFPS*.

Bizjak, Aleš, Gratzer, Daniel, Krebbers, Robbert, & Birkedal, Lars. (2018). *Iron: Managing obligations in higher-order concurrent separation logic*. Draft.

Bornat, Richard, Calcagno, Cristiano, O'Hearn, Peter W., & Parkinson, Matthew J. (2005). Permission accounting in separation logic. *Pages 259–270 of: POPL*.

Boyland, John. (2003). Checking interference with fractional permissions. *Pages 55–72 of: SAS*. LNCS, vol. 2694.

Brookes, Stephen. (2007). A semantics for concurrent separation logic. *TCS*, **375**(1–3), 227–270.

Buisse, Alexandre, Birkedal, Lars, & Støvring, Kristian. (2011). Step-indexed Kripke model of separation logic for storable locks. *ENTCS*, **276**, 121–143.

Cao, Qinxiang, Cuellar, Santiago, & Appel, Andrew W. (2017). Bringing order to the separation logic jungle. *Pages 190–211 of: APLAS*. LNCS, vol. 10695.

Cohen, Ernie, Alkassar, Eyad, Boyarinov, Vladimir, Dahlweid, Markus, Degenbaev, Ulan, Hillebrand, Mark, Langenstein, Bruno, Leinenbach, Dirk, Moskal, Michał, Obua, Steven, Paul, Wolfgang, Pentchev, Hristo, Petrova, Elena, Santen, Thomas, Schirmer, Norbert, Schmaltz, Sabine, Schulte, Wolfram, Shadrin, Andrey, Tobies, Stephan, Tsyban, Alexandra, & Tverdyshev, Sergey. (2009). Invariants, modularity, and rights. *Pages 43–55 of: PSI*. LNCS, vol. 5947.

da Rocha Pinto, Pedro, Dinsdale-Young, Thomas, & Gardner, Philippa. (2014). TaDA: A logic for time and data abstraction. *Pages 207–231 of: ECOOP*. LNCS, vol. 8586.

Di Gianantonio, Pietro, & Miculan, Marino. (2002). A unifying approach to recursive and co-recursive definitions. *Pages 148–161 of: TYPES*. LNCS, vol. 2646.

Dijkstra, Edsger W. (1975). Guarded commands, nondeterminacy and formal derivation of programs. *CACM*, **18**(8), 453–457.

Dinsdale-Young, Thomas, Gardner, Philippa, & Wheelhouse, Mark J. (2010a). Abstraction and refinement for local reasoning. *Pages 199–215 of: VSTTE*. LNCS, vol. 6217.

Dinsdale-Young, Thomas, Dodds, Mike, Gardner, Philippa, Parkinson, Matthew J., & Vafeiadis, Viktor. (2010b). Concurrent abstract predicates. *Pages 504–528 of: ECOOP*. LNCS, vol. 6183.

Dinsdale-Young, Thomas, Birkedal, Lars, Gardner, Philippa, Parkinson, Matthew J., & Yang, Hongseok. (2013). Views: Compositional reasoning for concurrent programs. *Pages 287–300 of: POPL*.

Dockins, Robert, Hobor, Aquinas, & Appel, Andrew W. (2009). A fresh look at separation algebras and share accounting. *Pages 161–177 of: APLAS*. LNCS, vol. 5904.

Dodds, Mike, Feng, Xinyu, Parkinson, Matthew J., & Vafeiadis, Viktor. (2009). Deny-guarantee reasoning. *Pages 363–377 of: ESOP*. LNCS, vol. 5502.

Dodds, Mike, Jagannathan, Suresh, Parkinson, Matthew J., Svendsen, Kasper, & Birkedal, Lars. (2016). Verifying custom synchronization constructs using higher-order separation logic. *TOPLAS*, **38**(2), 4:1–4:72.

Dreyer, Derek, Neis, Georg, Rossberg, Andreas, & Birkedal, Lars. (2010). A relational modal logic for higher-order stateful ADTs. *Pages 185–198 of: POPL*.

Feng, Xinyu. (2009). Local rely-guarantee reasoning. *Pages 315–327 of: POPL*.

Feng, Xinyu, Ferreira, Rodrigo, & Shao, Zhong. (2007). On the relationship between concurrent separation logic and assume-guarantee reasoning. *Pages 173–188 of: ESOP*. LNCS, vol. 4421.

Frumin, Dan, Krebbers, Robbert, & Birkedal, Lars. (2018). ReLoC: A mechanised relational logic for fine-grained concurrency. *Pages 442–451 of: LICS*.

Fu, Ming, Li, Yong, Feng, Xinyu, Shao, Zhong, & Zhang, Yu. (2010). Reasoning about optimistic concurrency using a program logic for history. *Pages 388–402 of: CONCUR*. LNCS, vol. 6269.

Garillot, François, Gonthier, Georges, Mahboubi, Assia, & Rideau, Laurence. (2009). Packaging mathematical structures. *Pages 327–342 of: TPHOLs*. LNCS, vol. 5674.

Gotsman, Alexey, Berdine, Josh, Cook, Byron, Rinetzky, Noam, & Sagiv, Mooly. (2007). Local reasoning about storable locks and threads. *Pages 19–37 of: APLAS*. LNCS, vol. 4807.

Hobor, Aquinas, Appel, Andrew W., & Zappa Nardelli, Francesco. (2008). Oracle semantics for concurrent separation logic. *Pages 353–367 of: ESOP*. LNCS, vol. 4960.

Hobor, Aquinas, Dockins, Robert, & Appel, Andrew W. (2010). A theory of indirection via approximation. *Pages 171–184 of: POPL*.

Iris Team. (2017). *The Iris documentation and Coq development*. Available on the Iris project website at: http://iris-project.org.

Ishtiaq, Samin S., & O'Hearn, Peter W. (2001). BI as an assertion language for mutable data structures. *Pages 14–26 of: POPL*.

Jensen, Jonas Braband, & Birkedal, Lars. (2012). Fictional separation logic. *Pages 377–396 of: ESOP*. LNCS, vol. 7211.

Jung, Ralf, Swasey, David, Sieczkowski, Filip, Svendsen, Kasper, Turon, Aaron, Birkedal, Lars, & Dreyer, Derek. (2015). Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. *Pages 637–650 of: POPL*.

Jung, Ralf, Krebbers, Robbert, Birkedal, Lars, & Dreyer, Derek. (2016). Higher-order ghost state. *Pages 256–269 of: ICFP*.

Jung, Ralf, Jourdan, Jacques-Henri, Krebbers, Robbert, & Dreyer, Derek. (2018). RustBelt: Securing the foundations of the Rust programming language. *PACMPL*, **2**(POPL), 66:1–66:34.

Kaiser, Jan-Oliver, Dang, Hoang-Hai, Dreyer, Derek, Lahav, Ori, & Vafeiadis, Viktor. (2017). Strong logic for weak memory: Reasoning about release-acquire consistency in Iris. *Pages 17:1–17:29 of: ECOOP*. LIPIcs, vol. 74.

Kock, Anders. (1970). Monads on symmetric monoidal closed categories. *Archiv der Mathematik*, **21**(1), 1–10.

Kock, Anders. (1972). Strong functors and monoidal monads. *Archiv der Mathematik*, **23**(1), 113–120.

Krebbers, Robbert, Jung, Ralf, Bizjak, Aleš, Jourdan, Jacques-Henri, Dreyer, Derek, & Birkedal, Lars. (2017a). The essence of higher-order concurrent separation logic. *Pages 696–723 of: ESOP*. LNCS, vol. 10201.

Krebbers, Robbert, Timany, Amin, & Birkedal, Lars. (2017b). Interactive proofs in higher-order concurrent separation logic. *Pages 205–217 of: POPL*.

Krebbers, Robbert, Jourdan, Jacques-Henri, Jung, Ralf, Tassarotti, Joseph, Kaiser, Jan-Oliver, Timany, Amin, Charguéraud, Arthur, & Dreyer, Derek. (2018). MoSeL: A general, extensible modal framework for interactive proofs in separation logic. *PACMPL*, **2**(ICFP), 77:1–16:30.

Kripke, Saul A. (1965). Semantical analysis of intuitionistic logic I. *Formal systems and recursive functions*, 92–130.

Krishnaswami, Neelakantan R., Turon, Aaron, Dreyer, Derek, & Garg, Deepak. (2012). Superficially substructural types. *Pages 41–54 of: ICFP*.

Krogh-Jespersen, Morten, Svendsen, Kasper, & Birkedal, Lars. (2017). A relational model of types-and-effects in higher-order concurrent separation logic. *Pages 218–231 of: POPL*.

Leino, K. Rustan M. (2010). Dafny: An automatic program verifier for functional correctness. *Pages 348–370 of: LPAR*. LNCS, vol. 6355.

Leino, K. Rustan M., Müller, Peter, & Smans, Jan. (2009). Verification of concurrent programs with Chalice. *Pages 195–222 of: FOSAD*. LNCS, vol. 5705.

Ley-Wild, Ruy, & Nanevski, Aleksandar. (2013). Subjective auxiliary state for coarse-grained concurrency. *Pages 561–574 of: POPL*.

Müller, Peter, Schwerhoff, Malte, & Summers, Alexander J. (2016). Viper: A verification infrastructure for permission-based reasoning. *Pages 41–62 of: VMCAI*. LNCS, vol. 9583.

Nakano, Hiroshi. (2000). A modality for recursion. *Pages 255–266 of: LICS*.

Nanevski, Aleksandar, Ley-Wild, Ruy, Sergey, Ilya, & Delbianco, Germán Andrés. (2014). Communicating state transition systems for fine-grained concurrent resources. *Pages 290–310 of: ESOP*. LNCS, vol. 8410.

O'Hearn, Peter W. (2007). Resources, concurrency, and local reasoning. *TCS*, **375**(1), 271–307.

O'Hearn, Peter W., & Pym, David J. (1999). The logic of bunched implications. *Bulletin of Symbolic Logic*, **5**(2), 215–244.

O'Hearn, Peter W., Reynolds, John C., & Yang, Hongseok. (2001). Local reasoning about programs that alter data structures. *Pages 1–18 of: CSL*. LNCS, vol. 2142.

Parkinson, Matthew J. (2010). The next 700 separation logics - (Invited paper). *Pages 169–182 of: VSTTE*. LNCS, vol. 6217.

Pilkiewicz, Alexandre, & Pottier, François. (2011). The essence of monotonic state. *Pages 73–86 of: TLDI*.

Pottier, François. (2013). Syntactic soundness proof of a type-and-capability system with hidden state. *JFP*, **23**(1), 38–144.

Reynolds, John C. (2000). Intuitionistic reasoning about shared mutable data structure. *Pages 303–321 of: Millennial Perspectives in Computer Science*.

Reynolds, John C. (2002). Separation logic: A logic for shared mutable data structures. *Pages 55–74 of: LICS*.

Sergey, Ilya, Nanevski, Aleksandar, & Banerjee, Anindya. (2015). Mechanized verification of fine-grained concurrent programs. *Pages 77–87 of: PLDI*.

Sozeau, Matthieu. (2009). A new look at generalized rewriting in type theory. *Journal of formalized reasoning*, **2**(1), 41–62.

Svendsen, Kasper, & Birkedal, Lars. (2014). Impredicative concurrent abstract predicates. *Pages 149–168 of: ESOP*. LNCS, vol. 8410.

Swasey, David, Garg, Deepak, & Dreyer, Derek. (2017). Robust and compositional verification of object capability patterns. *PACMPL*, **1**(OOPSLA), 89:1–89:26.

Tassarotti, Joseph, & Harper, Robert. (2018). *A separation logic for concurrent randomized programs*. Draft.

Tassarotti, Joseph, Jung, Ralf, & Harper, Robert. (2017). A higher-order logic for concurrent termination-preserving refinement. *Pages 909–936 of: ESOP*. LNCS, vol. 10201.

Timany, Amin, & Birkedal, Lars. (2018). *Mechanized relational verification of concurrent programs with continuations*. Draft.

Timany, Amin, Stefanesco, Léo, Krogh-Jespersen, Morten, & Birkedal, Lars. (2018). A logical relation for monadic encapsulation of state: Proving contextual equivalences in the presence of `runST`. *PACMPL*, **2**(POPL), 64:1–64:28.

Turon, Aaron, Dreyer, Derek, & Birkedal, Lars. (2013). Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. *Pages 377–390 of: ICFP*.

Turon, Aaron, Vafeiadis, Viktor, & Dreyer, Derek. (2014). GPS: Navigating weak memory with ghosts, protocols, and separation. *Pages 691–707 of: OOPSLA*.

Vafeiadis, Viktor, & Parkinson, Matthew J. (2007). A marriage of rely/guarantee and separation logic. *Pages 256–271 of: CONCUR*. LNCS, vol. 4703.

Wildmoser, Martin, & Nipkow, Tobias. (2004). Certifying machine code safety: Shallow versus deep embedding. *Pages 305–320 of: TPHOLs*. LNCS, vol. 3223.