Higher-Order Ghost State

Ralf Jung

MPI-SWS, Germany

jung@mpi-sws.org

Robbert Krebbers

Aarhus University, Denmark

mail@robbertkrebbers.nl

Lars Birkedal

Aarhus University, Denmark

birkedal@cs.au.dk

Derek Dreyer

MPI-SWS, Germany

dreyer@mpi-sws.org

Abstract

The development of *concurrent separation logic* (CSL) has sparked a long line of work on modular verification of sophisticated concurrent programs. Two of the most important features supported by several existing extensions to CSL are *higher-order quantification* and *custom ghost state*. However, none of the logics that support both of these features reap the full potential of their combination. In particular, none of them provide general support for a feature we dub *"higher-order ghost state"*: the ability to store arbitrary higher-order separation-logic predicates in ghost variables.

In this paper, we propose higher-order ghost state as a interesting and useful extension to CSL, which we formalize in the framework of Jung *et al.*'s recently developed Iris logic. To justify its soundness, we develop a novel algebraic structure called CMRAs ("cameras"), which can be thought of as "step-indexed partial commutative monoids". Finally, we show that Iris proofs utilizing higher-order ghost state can be effectively formalized in Coq, and discuss the challenges we faced in formalizing them.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory;   F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords Separation logic, fine-grained concurrency, higher-order logic, compositional verification, interactive theorem proving

1. Introduction

Over a decade ago, O'Hearn made a critical observation: separation logic—developed to simplify the verification of sequential, heap-manipulating programs—can help simplify the verification of concurrent programs as well. In *concurrent separation logic* (CSL) [28], assertions denote not only facts about the state of the program, but also *ownership* of a piece of that state. Concretely, this means that if a thread t can assert $\ell \mapsto v$, then t knows not only that location ℓ currently points to v, but also that it "owns" ℓ, so no other thread can read or write ℓ concurrently. Given this ownership assertion, t can perform *local* (and essentially sequential) reasoning on accesses to ℓ, completely ignoring concurrently operating threads.

Of course at some point threads have to communicate through some kind of shared state (such as a mutable heap or message-passing channels). To reason modularly about such communication, the original CSL used a simple form of resource invariants, which

were tied to a "conditional critical region" construct for synchronization. Since O'Hearn's pioneering (and Gödel-award-winning) paper, there has been an avalanche of follow-on work extending CSL with more sophisticated mechanisms for modular reasoning, which allow shared state to be accessed at a finer granularity (*e.g.,* atomic compare-and-swap instructions) and which support the verification of more "daring" (less clearly synchronized) concurrent programs [40, 17, 16, 13, 18, 38, 35, 27, 11, 24].

In this paper, we focus on two of the most important extensions to CSL—*higher-order quantification* and *custom ghost state*—and observe that, although several logics support both of these extensions, none of them reap the full potential of their combination. In particular, none of them provide general support for a feature we dub "higher-order ghost state".

Higher-order quantification is the ability to quantify logical assertions (universally and existentially) over other assertions and, in general, over arbitrary higher-order predicates. Several recent extensions to CSL have incorporated higher-order quantification [36, 35, 24, 21, 27], in part because it leads to more generic and reusable specifications of concurrent data structures (see §4), and in part because it is seemingly necessary for verifying some higher-order concurrency paradigms [35, 38, 31].

Ghost state is "logical state", *i.e.,* state that is essential to maintain in the *proof* of a program but is not part of the physical state manipulated by the program itself. It is a fixture of Hoare logics since the work of Owicki and Gries [29] in the 1970s, and is useful for a variety of purposes: for encoding various kinds of "permissions", for recording information about the trace of the computation, for describing "protocols" on how threads may interact with shared state, and more. Traditionally, ghost state was manipulated by instrumenting a program with updates to "ghost" (or "auxiliary") variables. Although this approach is convenient for integration into automatic verification tools [10], it is unnecessarily low-level: there is no reason logical state needs to be manipulated in exactly the same way as physical state, and doing so makes it harder to reason about updates to shared logical state in a modular fashion.

Recently, a number of researchers have argued that a more high-level, general, and flexible way to represent ghost state is via *partial commutative monoids (PCMs)*. Intuitively, PCMs are a natural fit for ghost state because they impose only the bare minimum requirements on something that should be "ownable" in a separation logic, while leaving lots of room for proof-specific customization. Several newer extensions to CSL [24, 27, 12] thus give users the freedom to define ghost state on a per-proof basis in terms of an arbitrary PCM of their choosing. Furthermore, the Iris logic [24] has established that PCMs (together with simple invariants) are flexible enough to derive several advanced reasoning mechanisms that were built in as primitive in prior logics.

Unfortunately, a limitation arises when one uses PCMs to support custom ghost state in the context of a logic with higher-order quantification. Specifically, PCMs yield a model of ghost state that is *first-order*. By this we mean that there is an inherent stratification:

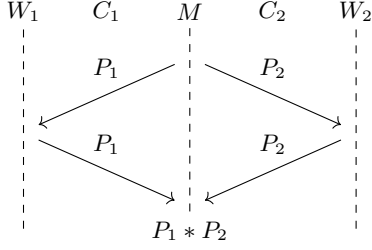$$\begin{array}{ccccc} W_1 & C_1 & M & C_2 & W_2 \end{array}$$

Figure 1. Structure of illustrative example

separation-logic assertions may talk about ownership of ghost state, but the PCM structure of ghost state may not depend (recursively) on the language of logical assertions. At first glance, this may not seem like a big deal. Why would you want *higher-order ghost state—i.e.,* ghost state that is mutually recursive with the language of assertions—anyway? What would that even mean?

To give some intuition for what higher-order ghost state is and what it can be good for, let us turn to a simple yet interesting illustrative example of some intuitively sensible reasoning that, as far as we know, no existing logic can support.

1.1 Illustrative Example: Splitting and Joining Existentials

Imagine we have three threads—a main thread M and two worker threads W_1 and W_2—running concurrently, and that we have two communication channels C_1 and C_2 connecting each worker to the main thread (see Figure 1). The idea of the example is that M is going to initialize some state, split up ownership of the state, send the pieces to W_1 and W_2 via their respective channels, and after some computation, they are going to send their pieces back so that M can join them together and reassert ownership of the whole state.

Formally: Let i range over $\{1, 2\}$. Let us suppose that M will initialize its state so that it satisfies the assertion P, after which M would like to split up ownership into two "disjoint" pieces P_1 and P_2 such that $P = P_1 * P_2$ and P_i is the assertion to be transferred back and forth with the corresponding W_i over C_i. (The $*$ operator here is the "separating conjunction" connective of separation logic, which enforces the disjointness of P_1 and P_2.) To make this work, it is essential that M and W_i agree ahead of time on the fact that C_i will be used to transfer ownership of P_i.

So far, so good. But now what happens if the assertion P depends on some information (say, on some x of type T) that is only determined *dynamically*, right before M wants to transfer ownership of P to the other, already running, threads. (For instance, x might be some information that depends on the result of I/O or some other source of nondeterminism.) In this case, we cannot associate the channels C_i ahead of time with the assertions P_i since, at the point the channels are created, the identity of x is not known. Instead, all we can really do is to existentially quantify over x—that is, set up our channels so that instead of transferring P_i to W_i, M will transfer $\exists x{:}T.\ P_i$, which *is* well-defined at the point the channels are created.

In the forward direction (sending from M to W_i), this works because $P = P_1 * P_2$ implies $\exists x{:}T.\ P \Rightarrow \exists x{:}T.\ P_1 * \exists x{:}T.\ P_2$. But when the W_i try to return the state to M, we run into a problem: although $P_1 * P_2 \Rightarrow P$, it is *not* true that $\exists x{:}T.\ P_1 * \exists x{:}T.\ P_2 \Rightarrow \exists x{:}T.\ P$ because there is no way to deduce that the existential witnesses for x in $\exists x{:}T.\ P_1$ and $\exists x{:}T.\ P_2$ are the same. This is all very frustrating since we know that, at least when M split up its original state, the witnesses were indeed the same.

Fortunately, there is another way: custom ghost state! Using custom ghost state, we can allocate a *oneshot* ghost variable γ up front: initially, γ will be undefined, and once M determines what

term e should be the witness for x, it will set γ to e.[1] We will write $\gamma \hookrightarrow e$ to denote the knowledge that e has been stored in γ. Unlike normal heap ownership assertions, $\gamma \hookrightarrow e$ can be freely duplicated (*i.e.,* $\gamma \hookrightarrow e \Leftrightarrow \gamma \hookrightarrow e * \gamma \hookrightarrow e$) since γ can only be assigned once. Furthermore, if we know $\gamma \hookrightarrow e_1$ and $\gamma \hookrightarrow e_2$, then we also know that $e_1 = e_2$ because, once set, γ maps to a unique e.

We can now amend the assertions $\exists x{:}T.\ P_i$ that are sent between the threads to $\exists x{:}T.\ (\gamma \hookrightarrow x * P_i)$, thus recording the fact that the witness x is precisely the term stored in γ. And the virtue of this encoding is that now the existential can be split *and* rejoined: $\exists x{:}T.\ (\gamma \hookrightarrow x * P) \Leftrightarrow \exists x{:}T.\ (\gamma \hookrightarrow x * P_1) * \exists x{:}T.\ (\gamma \hookrightarrow x * P_2)$. The $\Rightarrow$ direction relies on the fact that $\gamma \hookrightarrow x$ is duplicable, and the $\Leftarrow$ direction relies on the fact that γ stores a unique witness.

There is just one catch: in all the existing logics that support custom ghost state, the witness x stored in γ is restricted to be *first-order* (*i.e.,* an "object-level" term), hence restricting all the existentials in the example to be first-order as well. If x instead is (or contains) an assertion or higher-order predicate of the logic—as may very well be the case if we are working in a higher-order concurrent separation logic [36, 35, 24, 27]—then the whole approach falls over. Moreover, this problem is not limited to logics with custom ghost state: to our knowledge, *no existing logic is capable of proving this example in the general case*.

The good news is that restricting custom ghost state to first-order is unnecessary. To overcome it, we will need the ability to store terms of arbitrary higher type T (including assertions and higher-order predicates) in ghost variables like γ. In other words, we will need *higher-order ghost state*.

1.2 Contributions

In this paper, we make the following contributions:

- We propose and formalize higher-order ghost state as an interesting and useful extension of concurrent separation logic.

- We give a semantics to justify this extension in terms of a novel algebraic structure we call a CMRA ("camera"), which can be thought of as a kind of "step-indexed" PCM.

- We demonstrate that separation-logic proofs utilizing higher-order ghost state can be effectively formalized in Coq.

To be concrete, we will develop our notion of higher-order ghost state as an extension to the **Iris** logic of Jung *et al.* [24]. Iris is a higher-order variant of CSL that supports custom ghost state through the choice of an arbitrary user-supplied PCM. Over the course of the paper, we will gradually extend Iris from its original version, numbered 1.0, to the latest version, numbered 2.0, which can handle higher-order ghost state in a generic manner.

In §2, we give an overview of the key features of Iris and showcase their use on a simple representative example. Along the way, we introduce Iris 1.1, which makes some improvements to the original Iris 1.0, in particular generalizing its model of resources from PCMs to *resource algebras (RAs)*, which are more flexible.

In §3, we extend Iris with higher-order ghost state and put the extension on a sound formal footing. This involves revisiting the category-theoretic model of Iris 1.x and further generalizing its model of resources to the aforementioned step-indexed CMRAs. Furthermore, we discuss how the new semantic structure of our higher-order ghost state not only generalizes the original model of Iris—in certain ways, it also simplifies it.

In §4, we return to our example from §1.1, flesh it out more carefully, and show how to verify it in Iris 2.0. Our implementation of the example makes use of a "barrier" synchronization primitive

---

[1] This is just one particular form of proof-specific ghost state that can be encoded as a PCM—we will see in §2.1 how to encode it.

(drawn from Dodds *et al.* [15]), whose own proof of correctness independently involves an interesting use of higher-order ghost state.

In §5, we describe our Coq formalization of Iris 2.0, which improves significantly on the original Coq formalization of Iris 1.0 in several respects. All proofs backing this paper have been formalized in Coq [1].

Finally, in §6 and §7, we conclude with related and future work.

2. Iris Primer

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 large variety of languages. For the purpose of this paper, we instantiate Iris with an ML-like language with higher-order store, fork, and compare-and-swap (`CAS`):

$$e ::= x \mid \texttt{rec}\, f(x) = e \mid e_1(e_2) \mid \texttt{assert}(e) \mid \texttt{fork}\ \{e\} \mid$$
$$\mid \texttt{ref}(e) \mid !e \mid e \leftarrow e \mid \texttt{CAS}(e, e_1, e_2) \mid \ldots$$

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

The logic includes the usual connectives and rules of higher-order separation assertion logic, some of which are shown in the grammar below.[2] In this section, we will give a brief tour of Iris, and demonstrate the purpose of these most important logical connectives.

$$P, Q, R ::= \textsf{True} \mid \textsf{False} \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q$$
$$\mid \forall x.\, P \mid \exists x.\, P \mid P * Q \mid \ell \mapsto v \mid \triangleright P \mid t = u$$
$$\mid \boxed{P}^{\iota} \mid \overline{\lfloor a \rfloor}^{\gamma} \mid \mathcal{V}(a) \mid \{P\}\, e\, \{v.\, Q\} \mid P \Rrightarrow Q \mid \ldots$$

What makes Iris a *higher-order* separation logic is that universal and existential quantifiers can range over any type, including that of assertions and (higher-order) predicates. Furthermore, notice that Hoare triples $\{P\}\, e\, \{v.\, Q\}$ are part of the assertion logic instead of being a separate entity. As a consequence, triples can be used in the same way as any logical assertion, and in particular, they can be nested to account for specifications of higher-order functions.

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

The function `mk_oneshot` allocates a oneshot location at *x* and returns a record with two functions. (Formally, records are syntactic sugar for pairs.) The function `tryset`(n) tries to set the location to n, which will fail if the location has already been set. We use `CAS` to ensure correctness of the check even if two threads concurrently try to set the location. The function `check`() records the current state of the location and then returns a closure which, if the location 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[3] succeed: 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 postconditions have 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 assertions, 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

---

[2] 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. For more details on this, see [24, 1].

[3] Our semantics here is that `assert`(e) gets stuck if e evaluates to `false`.

```
mk_oneshot ≜ λ_. let x = ref(inl(0)) in
    { tryset = λn. CAS(x, inl(0), inr(n)),
      check = λ_. let y = !x in λ_.
                  match y, !x with
                    inl(_) , _      ⇒ ()
                  | inr(n), inl(_) ⇒ assert(false)
                  | inr(n), inr(m) ⇒ assert(n = m)
                  end }
```

$\{\textsf{True}\}\ \texttt{mk\_oneshot}()$
$$\left\{ \begin{array}{l} c.\ \forall v.\ \{\textsf{True}\}\ c.\texttt{tryset}(v)\ \{w.\ w \in \{\textsf{true}, \textsf{false}\}\}\ * \\ \quad \{\textsf{True}\}\ c.\texttt{check}()\ \{f.\ \{\textsf{True}\}\ f()\ \{\textsf{True}\}\} \end{array} \right\}$$

Figure 2. Example code and specification

---

logic, the specification for `mk_oneshot` actually permits the client to call `tryset` and `check`, as well as the f returned by `check`, concurrently from multiple threads and in any combination.

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 γ 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 reason we do this is so that we can control what kind of *sharing* is possible on the location. For a physical location ℓ, the assertion $\ell \mapsto v$ 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* 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*.

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

2.1 Ghost State in Iris: Resource Algebras

The key properties of ownership of ghost state in concurrent separation logics are:

- Ownership of different threads can be composed.

- Composition of ownership is associative and commutative, mirroring the associative and commutative semantics of parallel composition.

- Combinations of ownership that do not make sense are ruled out, *e.g.,* multiple threads claiming to have exclusive ownership of the same piece of ghost state.

For these reasons, *partial commutative monoids* (PCMs) have become a canonical structure for representing ghost state in separation logics. Iris 1.0 was parameterized by an arbitrary PCM, so that the structure of the ghost state was entirely up to the user.

In Iris 1.1, we are deviating slightly from this, using our own notion of a *resource algebra* (RA), whose definition is in Figure 3. As we will see in our example, the additional flexibility afforded by RAs results in additional logical expressiveness.

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

$$\forall a, b, c.\ (a \cdot b) \cdot c = a \cdot (b \cdot c) \qquad \text{(RA-ASSOC)}$$

$$\forall a, b.\ a \cdot b = b \cdot a \qquad \text{(RA-COMM)}$$

$$\forall a.\ |a| \in M \Rightarrow |a| \cdot a = a \qquad \text{(RA-CORE-ID)}$$

$$\forall a.\ |a| \in M \Rightarrow ||a|| = |a| \qquad \text{(RA-CORE-IDEM)}$$

$$\forall a, b.\ |a| \in M \wedge a \preccurlyeq b \Rightarrow |b| \in M \wedge |a| \preccurlyeq |b|$$
$$\text{(RA-CORE-MONO)}$$

$$\forall a, b.\ (a \cdot b) \in \mathcal{V} \Rightarrow a \in \mathcal{V} \qquad \text{(RA-VALID-OP)}$$

where $\quad M^? \triangleq M \uplus \{\varepsilon\} \qquad a^? \cdot \varepsilon \triangleq \varepsilon \cdot a^? \triangleq a^?$

$$a \preccurlyeq b \triangleq \exists c \in M.\ b = a \cdot c \qquad \text{(RA-INCL)}$$

Figure 3. Resource algebras

There are two key differences between RAs and PCMs:

1. The composition operation on RAs is total (as opposed to the partial composition operation of a PCM), but there is a specific subset $\mathcal{V}$ of *valid* elements that is compatible with the composition operation (RA-VALID-OP).

   We will see in §3.2 that this take on partiality is necessary when defining the structure of *higher-order* ghost state.

2. Instead of a single unit that is an identity to every element, we allow for an arbitrary number of units, via a function $|-|$ assigning 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).

   Notice that the domain of the core is $M^?$, a set that adds a dummy element ε to M. Thus, the core can be *partial*: not all elements need to have a unit. We use the metavariable $a^?$ to indicate elements of $M^?$. We also lift the composition ($\cdot$) to $M^?$. As we will see in §2.5, partial cores help us to build interesting composite RAs from smaller primitives.

   Notice also that the core of an RA is a strict generalization of the unit that any PCM must provide, since $|-|$ can always be picked as a constant function.

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. This RA appropriately mirrors the state of the physical location x. The carrier is defined using a datatype-like notation as follows:

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

$$\mathcal{V} \triangleq \{\mathsf{pending}\} \cup \{\mathsf{shot}(n) \mid n \in \mathbb{Z}\}$$

The two important states of the ghost location are: $\mathsf{pending}$, 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 $\bot$ to account for partiality.

The most interesting piece of data of an RA is of course its composition: What happens when ownership of two threads is combined? For our example, we are most interested in the behavior of $\mathsf{pending}$ and shot:

$$\mathsf{pending} \cdot \mathsf{pending} \triangleq \bot$$

$$\mathsf{pending} \cdot \mathsf{shot}(n) \triangleq \mathsf{shot}(n) \cdot \mathsf{pending} \triangleq \bot$$

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

$$\frac{\text{GHOST-OP}}{\boxed{a \cdot b}^\gamma \Leftrightarrow \boxed{a}^\gamma * \boxed{b}^\gamma} \qquad \frac{\text{GHOST-VALID}}{\boxed{a}^\gamma \Rightarrow \boxed{a}^\gamma * \mathcal{V}(a)}$$

$$\frac{\text{GHOST-ALLOC}}{\mathsf{True} \Rrightarrow \exists \gamma.\ \boxed{a}^\gamma} \qquad \frac{\text{GHOST-UPDATE}}{\boxed{a}^\gamma \Rrightarrow \boxed{b}^\gamma} \qquad \frac{\text{PERSISTENT-GHOST}}{\mathsf{persistent}(\boxed{a}^\gamma)}$$

$$\text{where } a \rightsquigarrow b \triangleq \forall a_\mathsf{f}^? \in M^?.\ a \cdot a_\mathsf{f}^? \in \mathcal{V} \Rightarrow b \cdot a_\mathsf{f}^? \in \mathcal{V}$$

$$\begin{array}{c} \text{INV} \\ \text{(side conditions on } \iota \text{ and } I \text{ omitted)} \\ \dfrac{\{I * P\}\ e\ \{v.\ I * Q\} \qquad \mathsf{atomic(e)}}{\boxed{I}^\iota \vdash \{P\}\ e\ \{v.\ Q\}} \end{array}$$

$$\frac{\text{INV-ALLOC}}{I \Rrightarrow \boxed{I}^\iota}$$

$$\frac{\text{CSQ}}{P \Rrightarrow P' \qquad \{P'\}\ e\ \{v.\ Q'\} \qquad \forall v.\ Q' \Rrightarrow Q}{\{P\}\ e\ \{v.\ Q\}}$$

$$\frac{\text{PERSISTENT-DUP}}{\mathsf{persistent}(P)}{P \Leftrightarrow P * P} \qquad \frac{\text{PERSISTENT-HOARE}}{Q \vdash \{P\}\ e\ \{v.\ R\}}{\{P * Q\}\ e\ \{v.\ R\}}\ \mathsf{persistent}(Q)$$

Figure 4. Some Iris 1.1 proof rules

As a result of this definition, if we *own* $\mathsf{pending}$, we know that no other thread can own anything about this location, since the composition with that other piece of ownership would be invalid (*i.e.,* $\bot$). Furthermore, if we own $\mathsf{shot}(n)$, we know that the *only* ownership any other thread can have is *also* $\mathsf{shot}(n)$, since everything else is incompatible with our own ownership. However, since $\mathsf{shot}(n) \cdot \mathsf{shot}(n) = \mathsf{shot}(n)$, we can also duplicate ownership of the location as much as we want, *once it has been set to some n*. This gives rise to the sharing that we need.

In defining the duplicable core, we make sure that we keep as much ownership as possible while satisfying RA-CORE-ID:

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

Note that, since ownership of $\mathsf{pending}$ is exclusive, it has no suitable unit element, so we assign it the "dummy core" ε. It is now straightforward to verify that the RA for our example satisfies the RA axioms.

Proof rules for ghost state. Resource algebras are embedded into the logic using the assertion $\boxed{a}^\gamma$, which asserts ownership of a piece a of the ghost location γ.[4] The proof rule GHOST-ALLOC in Figure 4 can be used to allocate a new ghost location, with an arbitrary but valid initial state a. The assertion $\mathcal{V}(a)$ reflects validity of a into the logic, *i.e.,* it corresponds to $a \in \mathcal{V}$ on the meta-level. The $\Rrightarrow$ in GHOST-ALLOC is a *view shift*, also known as a *ghost move*: $P \Rrightarrow Q$ says that, starting in a (ghost and physical) state satisfying P, we can do updates *to the ghost state only* and arrive in a state satisfying Q. The rule of consequence (CSQ) says that we can apply view shifts in the pre- and postconditions of Hoare triples.

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.

Finally, we need a way to update ghost locations. Updates to ghost locations are called *frame-preserving updates* and can be

---

[4] The notation $\gamma \hookrightarrow e$ used in §1.1 is defined in terms of the more general $\boxed{a}^\gamma$ (see §4.2).

performed using the rule GHOST-UPDATE. We can perform a frame-preserving update $a \rightsquigarrow b$ on a ghost location γ, if the update ensures that no matter what assumptions the rest of the program is making about the state of γ, if these assumptions were compatible with a, they should also be compatible with b. Or in other words, we have to make sure that the assumptions the rest of the program is making about γ will not be invalidated by the update.

We only need one frame-preserving update for our example:

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

This property follows from the fact that no frames are compatible with $\mathsf{pending}$. That is, by the definition of $(\cdot)$, we know that $\mathsf{pending} \cdot a_{\mathrm{f}}^? \in \mathcal{V}$ can only hold for the dummy frame $a_{\mathrm{f}}^? = \varepsilon$.

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 x. This is done using an *invariant*.

In Iris, invariants are provided in the form of the assertion $\boxed{P}^{\iota}$, which asserts that P is maintained as an invariant on the global (ghost and physical) state. In order to identify invariants, every invariant has a name ι. We need to do some bookkeeping with these names to avoid *reentrancy*, which in the case of invariants means avoiding "opening" the same invariant twice in a nested fashion. However, we omit the bookkeeping for this paper, as it is orthogonal to our focus; see the original Iris paper for details [24].

Notice that $\boxed{P}^{\iota}$ is *just another kind of assertion*, and can be used anywhere that normal assertions can be used—including the pre- and postconditions of Hoare triples, *and invariants themselves*. The latter property is sometimes referred to as *impredicativity*.

Invariants are created using the INV-ALLOC rule (Figure 4): Whenever an assertion P has been established, it can be turned into an invariant. Allocating an invariant is another example of a view shift, since only ghost state is changed to record that this invariant will henceforth be maintained.

2.3 Persistent Assertions

Before we come to the actual proof of Figure 2, we have to talk about the notion of a *persistent* assertion. These are assertions that, once established, will remain valid for the rest of the verification. Examples of persistent assertions are invariants $\boxed{P}^{\iota}$, validity $\mathcal{V}(a)$, equality $t = u$, Hoare triples $\{P\}\, e\, \{v.\, Q\}$ and view shifts $P \Rrightarrow Q$. Persistent assertions can be freely duplicated (PERSISTENT-DUP) and can be moved in and out of the precondition of Hoare triples (PERSISTENT-HOARE). This differentiates them from *ephemeral* assertions like $\ell \mapsto v$ and $\lfloor a \rfloor^{\gamma}$, which could be invalidated in the future by actions of the program or the proof, and for which the above properties thus do not hold.

Another example of a persistent assertion is *ghost ownership of a core*: $\lfloor |a| \rfloor^{\gamma}$ is persistent (PERSISTENT-GHOST). This possibility of having *persistent ghost ownership* is a novel concept of Iris 1.1 that cannot be expressed in a PCM-based logic. In the proof of check, we will see this concept in action.

A closely related concept is the notion of *duplicable assertions*, *i.e.,* assertions P for which one has $P \Leftrightarrow P * P$. This is a strictly weaker notion, however: not all duplicable assertions are persistent. For example, $\exists q.\, \ell \xmapsto{q} v$ is duplicable (which follows from halving the fractional permission q), but is not persistent.

2.4 Proof of the Example

Proof of `mk_oneshot`. In order to verify `mk_oneshot`, we will first allocate a new ghost location γ with the structure of the oneshot RA defined above, picking the initial state $\mathsf{pending}$. Then, we will establish and allocate the following invariant:

$$I \triangleq (x \mapsto \mathtt{inl}(0) * \lfloor \mathsf{pending} \rfloor^{\gamma}) \vee (\exists n.\, x \mapsto \mathtt{inr}(n) * \lfloor \mathsf{shot}(n) \rfloor^{\gamma})$$

Since x is initialized with $\mathtt{inl}(0)$, the invariant I initially holds.

Proof of `tryset`. How exactly can invariants be used? This is described by INV, which allows us to *open* the invariant for the verification of e. That is, we can assume that the invariant holds in the precondition, and then we have to reestablish that it holds in the postcondition. Crucially, we require that e is *physically atomic*, meaning that it is guaranteed to evaluate to a value in a single step of computation. This side condition is essential for soundness: within the verification of e the invariant I might be temporarily broken, but by restricting e's execution to a single instruction we ensure that no other thread can observe that I has been broken. In our language, reading from memory ($!e$), assigning to memory ($e_1 \leftarrow e_2$), and `CAS` are all physically atomic operations.

In the case of `tryset`, we have to open the invariant to justify safety of the `CAS`. The invariant always provides $x \mapsto \_$, 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 $\mathtt{inl}(0)$, so we obtain the invariant in its left disjunct. Thus, after the `CAS`, we have the following:

$$x \mapsto \mathtt{inr}(n) * \lfloor \mathsf{pending} \rfloor^{\gamma}$$

How can we reestablish the invariant I after this `CAS`? Clearly, we must pick the right disjunct, since $x \mapsto \mathtt{inr}(n)$. 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 $\mathsf{pending}$. 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 $\mathtt{inr}(n)$.

Proof of `check`. What remains is to prove correctness of `check`. We open our invariant I to justify 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 $\mathtt{inr}(n)$, then x will not change its value. At this point in the proof, we have the following assertion:

$$x \mapsto y * ((y = \mathtt{inl}(0) * \lfloor \mathsf{pending} \rfloor^{\gamma}) \vee$$
$$(\exists n.\, y = \mathtt{inr}(n) * \lfloor \mathsf{shot}(n) \rfloor^{\gamma}))$$

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

$$x \mapsto y * ((y = \mathtt{inl}(0) * \lfloor \mathsf{pending} \rfloor^{\gamma}) \vee$$
$$(\exists n.\, y = \mathtt{inr}(n) * \lfloor \mathsf{shot}(n) \rfloor^{\gamma} * \lfloor \mathsf{shot}(n) \rfloor^{\gamma}))$$

which in turn implies:

$$I * \underbrace{(y = \mathtt{inl}(0) \vee (\exists n.\, y = \mathtt{inr}(n) * \lfloor \mathsf{shot}(n) \rfloor^{\gamma}))}_{P}$$

We can then reestablish the invariant, 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 justify that the assertion will hold.

To do so, we have to show that P is persistent. Technically, this is mandated by PERSISTENT-HOARE; intuitively, it is needed because the client could call the closure returned by `check` in an arbitrary future state of the program, and we have to make sure that it is always safe to do so. The reason P is persistent is that, thanks to $\mathsf{shot}(n) = |\mathsf{shot}(n)|$ and PERSISTENT-GHOST, $\lfloor \mathsf{shot}(n) \rfloor^{\gamma}$ is persistent. 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 safety of !x. Our assertion then is $I * P$, which unfolds to:

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

In order to proceed, we do 4-way case distinction over the values of x and y. Our goal is to prove that the assertions do not fail, so we have to show the following:

1. **Goal:** It cannot be the case that $y = \mathtt{inr}(n)$ and $x \mapsto \mathtt{inl}(\_)$. In this case, we would have $\boxed{\mathtt{pending} \cdot \mathtt{shot}(n)}^\gamma$ and hence $\boxed{\bot}^\gamma$, which according to GHOST-VALID is a contradiction.

2. **Goal:** In the case that $y = \mathtt{inr}(n)$ and $x \mapsto \mathtt{inr}(m)$, we have $\boxed{\mathtt{shot}(n) \cdot \mathtt{shot}(m)}^\gamma$. The following lemma shows that the assertion cannot fail:

$$\boxed{\mathtt{shot}(n) \cdot \mathtt{shot}(m)}^\gamma \Rightarrow n = m \qquad \text{(ONESHOT-AGREE)}$$

The lemma holds by GHOST-VALID, which yields $\mathcal{V}(\mathtt{shot}(n) \cdot \mathtt{shot}(m))$, implying $\mathtt{shot}(n) \cdot \mathtt{shot}(m) \neq \bot$, and thus $n = m$.

2.5 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, so 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 (ONESHOT-AGREE).

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 $\bot$.)

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

$$M_1 +_\bot M_2 \triangleq \mathtt{inl}(a_1 : M_1) \mid \mathtt{inr}(a_2 : M_2) \mid \bot$$
$$\mathcal{V} \triangleq \left\{ \mathtt{inl}(a_1) \,\middle|\, a_1 \in \mathcal{V}' \right\} \cup \left\{ \mathtt{inr}(a_2) \,\middle|\, a_2 \in \mathcal{V}'' \right\}$$
$$\mathtt{inl}(a_1) \cdot \mathtt{inl}(b_1) \triangleq \mathtt{inl}(a_1 \cdot b_1)$$
$$|\mathtt{inl}(a_1)| \triangleq \begin{cases} \varepsilon & \text{if } |a_1| = \varepsilon \\ \mathtt{inl}(|a_1|) & \text{otherwise} \end{cases}$$

The composition and core for $\mathtt{inr}$ are defined symmetrically. Above, $\mathcal{V}'$ refers to the validity of M_1, and $\mathcal{V}''$ to the validity of M_2.

Exclusive. Given a set X, the task of the *exclusive RA* $\mathrm{Ex}(X)$ is to make sure that one party *exclusively* owns a value $x \in X$. We define Ex by essentially dropping $\mathtt{shot}$ from the oneshot RA, and generalizing the carrier:

$$\mathrm{Ex}(X) \triangleq \mathtt{ex}(x : X) \mid \bot$$
$$\mathcal{V} \triangleq \{ \mathtt{ex}(x) \mid x \in X \}$$
$$|\mathtt{ex}(x)| \triangleq \varepsilon$$

Composition is always $\bot$ to ensure that ownership is exclusive.

$$\boxed{a \cdot b} \Leftrightarrow \boxed{a} * \boxed{b} \qquad\qquad \boxed{a} \Rightarrow \mathcal{V}(a) \qquad\qquad \frac{a \rightsquigarrow b}{\boxed{a} \Rrightarrow \boxed{b}}$$

Figure 5. Primitive rules for ghost state

Agreement. Given a set X, the task of the *agreement RA* $\mathrm{AG}(X)$ is to make sure multiple parties can *agree* upon which value $x \in X$ has been picked. We define AG by essentially dropping $\mathtt{pending}$ from the oneshot RA and generalizing the carrier:

$$\mathrm{AG}(X) \triangleq \mathtt{ag}(x : X) \mid \bot$$
$$\mathcal{V} \triangleq \{ \mathtt{ag}(x) \mid x \in X \}$$
$$\mathtt{ag}(x) \cdot \mathtt{ag}(y) \triangleq \begin{cases} \mathtt{ag}(x) & \text{if } x = y \\ \bot & \text{otherwise} \end{cases}$$
$$|\mathtt{ag}(x)| \triangleq \mathtt{ag}(x)$$

Oneshot. We can now define the general idea of the oneshot RA as $\mathrm{ONESHOT}(M) \triangleq \mathrm{Ex}(1) +_\bot M$, and recover the RA for the example as $\mathrm{ONESHOT}(\mathrm{AG}(\mathbb{Z}))$. Notice that the decomposition of ONESHOT into a sum relies crucially on the fact that one summand, namely $\mathrm{Ex}(1)$, has a partial core. The following generalization of ONESHOT-SHOOT can be shown in general for ONESHOT:

$$\mathtt{inl}(\mathtt{ex}()) \rightsquigarrow \mathtt{inr}(a)$$

This update relies on the fact that $\mathtt{ex}()$ has *no* frame. If there was a unit $|\mathtt{ex}()|$ for $\mathtt{ex}()$, then $\mathtt{inl}(|\mathtt{ex}()|)$ would be a frame compatible with $\mathtt{inl}(\mathtt{ex}())$, and the frame-preserving update would not hold. Obtaining usable frame-preserving updates for sums is one of our key motivations for making the core a partial function.

Joining existentials. The general construction $\mathrm{ONESHOT}(-)$ comes in handy, because it is exactly the RA that we need for the challenge of splitting and joining existentials as described in §1.1. Remember that there, too, we want to allocate ghost state before it is known what will be stored there.

Now, it may seem that we are already equipped to solve the challenge posed in §1.1, but that is not the case. To prove that example, we need the ghost state to be $M \triangleq \mathrm{ONESHOT}(\mathrm{AG}(T))$, where T may depend on *iProp* (the type of Iris assertions). However, the structure of the ghost state is a *parameter* of Iris, so *iProp* is actually defined in terms of M—which means we cannot define M in terms of *iProp*. We have to do some interesting work in the model of Iris to resolve this cycle, as we will describe in §3.

2.6 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 [24]. For example, both Hoare triples and assertions of the form $l \mapsto v$ are actually derived forms. 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 assertion $\boxed{a}^\gamma$ for ghost ownership, which is not a baked-in notion. Instead, the idea of having a *heap* of ghost variables, which can be individually allocated and updated, is derived within the logic. Although this construction did not fundamentally change since Iris 1.0 [24], it is important for understanding the model construction in §3.

As a primitive, Iris provides the construct $\boxed{a}$, which asserts ownership of a piece a of the *entire* global ghost state, rather than ownership of a piece of an individual ghost variable. The structure of the global ghost state is a *single* RA picked by the user, and the rules governing global ghost state are given in Figure 5.

Objects: tuples $(T, (\overset{n}{=} \subseteq T \times T)_{n\in\mathbb{N}})$ satisfying:

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

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

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

(Completeness axiom omitted)

Arrows: *non-expansive functions* $f : T \xrightarrow{\text{ne}} U$ satisfying:

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

Figure 6. Objects and arrows of the category $\mathcal{COFE}$ of COFEs

Step-indexed Assertions:

$$SProp \triangleq \{X \in \wp(\mathbb{N}) \mid \forall n \geq m.\ n \in X \Rightarrow m \in X\}$$

$$X \overset{n}{=} Y \triangleq \forall m \leq n.\ m \in X \Leftrightarrow m \in Y$$

Discrete: $\quad \Delta X \triangleq X$

$$x \overset{n}{=} y \triangleq x = y$$

Later: $\quad \blacktriangleright T \triangleq \{\text{next}(x) \mid x \in T\}$

$$\text{next}(x) \overset{n}{=} \text{next}(y) \triangleq n = 0 \lor x \overset{n-1}{=} y$$

Figure 7. The COFEs used as part of the Iris model

In practice, however, the end-user typically wants to use *multiple* ghost variables of *multiple* RAs, in particular when combining different proofs. We apply a general construction facilitating this: We assume a family of RAs $(M_i)_{i\in\mathcal{I}}$ for some index set $\mathcal{I}$, and then 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 point-wise 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.

This allows us (a) to use all the M_i in our proofs, and (b) to treat ghost state as a heap, where we can allocate new instances of any of the M_i at any time. We define local ghost ownership of a single location as:

$$\overline{a : M_i}^\gamma \triangleq \lambda j.\ \begin{cases} [\gamma \mapsto a] & \text{if } i = j \\ \emptyset & \text{otherwise} \end{cases}$$

In other words, $\overline{a : M_i}^\gamma$ 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 $\overline{a}^\gamma$. The rules given in Figure 4 can then be derived from those shown in Figure 5.

3. A Model for Higher-Order Ghost State

In this section, we explain how to solve the problem outlined at the end of §2.5 by changing the semantic model of Iris assertions to account for higher-order ghost state. We start by giving a model for Iris 1.1. This is essentially a review of how the model of Iris 1.0 was built; the differences are minuscule. Next we show why the agreement construction AG as defined in §2.5 does not scale to higher-order ghost state, and give a refined version that does scale. Finally, we show how to evolve the model to Iris 2.0 in order to solve the challenge posed in §1.1 in a generic and foundational way.

3.1 The Iris Model

Proving soundness of Iris involves giving a model to all its primitive assertions in an appropriate semantic domain, and then verifying correctness of every primitive rule with respect to that model.

The main difficulty here is to come up with a semantic domain that is sufficient to model Iris's assertions. This domain, which we call *iProp*, should satisfy (roughly) the following equations:

$$iProp = World \xrightarrow{\text{mon}} Res \xrightarrow{\text{mon}} Prop \qquad \text{(IRIS-1.0)}$$

where $\quad World \triangleq \mathbb{N} \xrightarrow{\text{fin}} iProp$

$$Res \triangleq M \times \text{Ex}(PhyState)$$

Let us look at the individual pieces: M is the RA of ghost state as picked by the user. Physical states *PhyState* are equipped with

the exclusive RA structure defined in §2.5, so that *Res* becomes the RA of resources (logical and physical) that can be owned. *Prop* is the domain of meta-level propositions (*e.g.,* Coq's `Prop`).

$Res \xrightarrow{\text{mon}} Prop$ is a predicate over resources, the usual model of assertions in separation logic. Iris is an *intuitionistic* separation logic, which gives rise to the monotonicity requirement—owning more resources makes more assertions true.[5] We can easily model ghost ownership $\overline{a}$ in this domain of resource predicates.

The *World* keeps track of the invariants that have been created. We index the resource predicate over "possible worlds". The invariant assertion $\boxed{P}^\iota$ is thus modeled as asserting the existence of an invariant named ι in the current world. Worlds are ordered by the extension order, *i.e.,* allocating new invariants "grows" the world. It is crucial to demand monotonicity of assertions with respect to that order, so that assertions cannot be invalidated by new invariants being added.

Notice that the above equations are circular: *World* depends on *iProp*, which depends contravariantly on *World*.

Step-indexing to the rescue. To solve the circularity in the proposed definition of *iProp*, we use *step-indexing* [3] to stratify the recursive domain equation. Following Birkedal *et al.* [7], we do this in an abstract way by working in the category $\mathcal{COFE}$ of *Complete Ordered Family of Equivalences* (COFEs), as defined in Figure 6.

The key intuition behind COFEs 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 (COFE-MONO). In the limit, it agrees with plain equality (COFE-LIMIT). In order to solve the recursive domain equation it is essential that COFEs are *complete, i.e.,* that any chain has a limit. See [7, 1] for more details.

An arrow $f : T \xrightarrow{\text{ne}} U$ between two COFEs is a ***non-expansive** function*. Such functions preserve the structure defined by $(\overset{n}{=})$ (COFE-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.

The COFEs that are essential to the Iris model are defined in Figure 7. The COFE of *step-indexed* assertions *SProp* represents assertions that are true for some number of steps, or forever. (*SProp* corresponds to the set of ordinals up to ω.) The *discrete COFE* ΔX turns any set X into a COFE by assigning the degenerate step-indexed equivalence. Finally, the *later* COFE $\blacktriangleright T$ moves the step-indexed equivalence of a COFE T up by one, thus making everything "one level more equal" than in T.

---

[5] In other words, the function has to be monotone with respect to the extension order $\preccurlyeq$ (defined in Figure 3) and implication on *Prop*.

It turns out that, within the category $\mathcal{COFE}$, the following recursive domain equation can be solved *up to isomorphism* using America and Rutten's theorem [2]:

$$iProp \approx World \xrightarrow{\text{mon}} \Delta Res \xrightarrow{\text{mon}} SProp \qquad (\text{IRIS-1.1})$$

$$\text{where} \quad World \triangleq \mathbb{N} \xrightarrow{\text{fin}} \blacktriangleright iProp$$

$$Res \triangleq M \times \text{Ex}(PhyState)$$

An Iris assertion is hence indexed by a world $w \in World$, by a resource $r \in Res$, and by a step-index $n \in \mathbb{N}$.

Compared to IRIS-1.0, the key differences are the use of *SProp* instead of *Prop*, and the $\blacktriangleright$ in front of the recursive occurrence of *iProp*. The $\blacktriangleright$ serves as a guard that introduces a stratification step. This is needed to ensure that the recursive domain equation reaches a fixed point. It is also the reason that the proof rule INV needs a side-condition about I: Due to *World* actually containing $\blacktriangleright iProp$ rather than *iProp*, INV only holds for assertions that do not depend on the step-index, such as ghost ownership $\lceil\overline{a}\rceil^\gamma$. We call such assertions *timeless*. There is a stronger version of INV [1] that works for any assertion, but it needs more care because we have to deal with the fact that I only holds at the *next* step-index (which is written as $\triangleright I$).

Since the equation IRIS-1.1 only holds up to isomorphism, we have to be careful not to perform operations that are not preserved by the isomorphism. Basically, we have to make sure that we stay *inside* the category $\mathcal{COFE}$, which in particular means that every function must be *non-expansive*.

However, the *end user* of Iris 1.1 does not have to bother much about this: Since we use the discrete COFE Δ for *Res*, all functions on *Res*—and in particular the RA operations for composition and core—are trivially non-expansive.

3.2 Higher-Order Agreement

With the framework of the model—in particular the category $\mathcal{COFE}$—set up, let us now look again at the agreement RA defined in §2.5. Remember that ultimately we would like to obtain $\text{AG}(T)$ for some T that may depend on *iProp*.

Ignoring the circularity that came up in §2.5, there is another problem with the definition of AG in that section: when the $\text{AG}(T)$ construction is generalized to range over a non-discrete COFE T, such as *iProp*, 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.*, $P \stackrel{n}{=} Q$ for some n while $P \neq Q$, we have:

$$\text{ag}(P) \cdot \text{ag}(Q) = \bot \quad \stackrel{n}{\neq} \quad \text{ag}(P) = \text{ag}(P) \cdot \text{ag}(P)$$

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

In order to make sense of the idea of agreement over COFEs, we have to drop absolute terms like the elements being *(exactly) equal* and composition being *(completely) defined*, and instead use *step-indexed* versions of those terms. This is achieved by keeping track of *how valid* an RA element is, which we can then relate to *how equal* the operands of the composition were. So, we re-define $\text{AG}(T)$ for any COFE T:

$$\text{AG}(T) \triangleq \{(x, V) \in T \times SProp\} / \sim$$

$$\text{where } a \sim b \triangleq a.V = b.V \land \forall n \in a.V.\ a.x \stackrel{n}{=} b.x$$

$$a \stackrel{n}{=} b \triangleq (\forall m \leq n.\ m \in a.V \Leftrightarrow m \in b.V) \land$$

$$(\forall m \leq n.\ m \in a.V \Rightarrow a.x \stackrel{m}{=} b.x)$$

$$\mathcal{V}_n \triangleq \{a \in \text{AG}(T) \mid n \in a.V\}$$

$$a \cdot b \triangleq \left(a.x, \left\{n \mid n \in a.V \land n \in b.V \land a \stackrel{n}{=} b\right\}\right)$$

A *CMRA* is a tuple $(M : \mathcal{COFE}, (\mathcal{V}_n \subseteq M)_{n \in \mathbb{N}},$ $|-| : M \xrightarrow{\text{ne}} M^?, (\cdot) : M \times M \xrightarrow{\text{ne}} M)$ satisfying:

$$\forall n, a, b.\ a \stackrel{n}{=} b \land a \in \mathcal{V}_n \Rightarrow b \in \mathcal{V}_n \quad (\text{CMRA-VALID-NE})$$

$$\forall n, m.\ n \geq m \Rightarrow \mathcal{V}_n \subseteq \mathcal{V}_m \quad (\text{CMRA-VALID-MONO})$$

$$\forall a, b, c.\ (a \cdot b) \cdot c = a \cdot (b \cdot c) \quad (\text{CMRA-ASSOC})$$

$$\forall a, b.\ a \cdot b = b \cdot a \quad (\text{CMRA-COMM})$$

$$\forall a.\ |a| \in M \Rightarrow |a| \cdot a = a \quad (\text{CMRA-CORE-ID})$$

$$\forall a.\ |a| \in M \Rightarrow ||a|| = |a| \quad (\text{CMRA-CORE-IDEM})$$

$$\forall a, b.\ |a| \in M \land a \preccurlyeq b \Rightarrow |b| \in M \land |a| \preccurlyeq |b| \quad (\text{CMRA-CORE-MONO})$$

$$\forall n, a, b.\ (a \cdot b) \in \mathcal{V}_n \Rightarrow a \in \mathcal{V}_n \quad (\text{CMRA-VALID-OP})$$

$$\forall n, a, b_1, b_2.\ a \in \mathcal{V}_n \land a \stackrel{n}{=} b_1 \cdot b_2 \Rightarrow$$

$$\exists c_1, c_2.\ a = c_1 \cdot c_2 \land c_1 \stackrel{n}{=} b_1 \land c_2 \stackrel{n}{=} b_2 \quad (\text{CMRA-EXTEND})$$

where

$$a \preccurlyeq b \triangleq \exists c.\ b = a \cdot c \quad (\text{CMRA-INCL})$$

Figure 8. CMRA operations and axioms

Elements $a \in \text{AG}(T)$ now consist of an $a.x \in T$ (the actual data represented by a) and a set $a.V$ defining *how valid* a is. The set contains those n such that a is valid for n steps of computation. Finally, the quotient by $\sim$ ensures that elements of $\text{AG}(T)$ are only equated for the number of steps for which they are valid.

In this construction, validity of a is generalized from a "plain" assertion $a \in \mathcal{V}$ to a step-indexed assertion $a \in \mathcal{V}_n$. Recall that validity in RAs corresponds to partiality in PCMs—we obtain a notion of two elements being composable or not. With step-indexed validity, we have a notion of elements being n-composable. You can think of this as "step-indexed partiality".

The step-indexed equality on $\text{AG}(T)$ propagates the one on T. This makes composition a non-expansive operation: For $a_1 \stackrel{n}{=} a_2$, it is easy to see that $a_1 \cdot b \stackrel{n}{=} a_2 \cdot b$ (and likewise if b changes).

In order to explain how we use AG, we first define an injection $\text{ag} : T \to \text{AG}(T)$ as follows:

$$\text{ag}(x) \triangleq (x, \mathbb{N})$$

Now imagine we start with two elements $x, y \in T$. You can see that $\text{ag}(x) \cdot \text{ag}(y)$ is n-valid if and only if $x \stackrel{n}{=} y$—in other words, the composition of two elements is as valid as the elements are equal. This gives rise to the following property:

$$\mathcal{V}(\text{ag}(x) \cdot \text{ag}(y)) \Rightarrow x = y \quad (\text{AG-VALID})$$

Note that this is a statement in Iris, not in the meta-logic. Both validity and equality are actually step-indexed assertions, namely, $\mathcal{V}_n$ and $(\stackrel{n}{=})$. This implication has to be shown once and for all when defining AG; it can then be used in Iris proofs.

There is one last technical step that we have to take: $\text{AG}(T)$ as defined in this section does not satisfy the completeness axiom of COFEs (see Figure 6). To turn $\text{AG}(T)$ into an actual COFE, we perform a Cauchy completeness construction (by considering chains of elements). This technical step is detailed in the appendix [1] and fully formalized in Coq, as are all the constructions in this paper.

3.3 CMRAs

The $\text{AG}(T)$ construction as defined in the previous section is no longer just an RA—we had to define $\mathcal{V}_n$, and we had to prove non-expansiveness of the operations. In this section we introduce the

algebraic structure of *CMRAs* ("cameras") to abstractly characterize such "step-indexed RAs". You can find its definition in Figure 8. The differences from RAs are as follows:

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

- The set of valid elements $\mathcal{V}_n$ is step-indexed. The rule CMRA-VALID-OP expresses that we need decomposition to preserve validity at *every* step-index. The rule CMRA-VALID-NE demands that $\mathcal{V}_n$ must not make a distinction between n-equal elements (this is a form of non-expansiveness), and CMRA-VALID-MONO declares that at smaller step-indices, more elements are valid.

  As discussed in the previous section, $\mathcal{V}_n$ encodes "step-indexed partiality", which is more general than using a partial function for composition. For consistency, we decided that partiality should be handled the same for RAs and CMRAs, which is why RAs use a validity predicate as well.

- The *extension* axiom CMRA-EXTEND roughly states that decomposition must commute with step-indexed equivalence. This is discussed in more depth in our technical appendix [1].

Notice that every RA can be trivially turned into a CMRA by equipping it with the discrete COFE Δ and making $\mathcal{V}_n$ just ignore the step-index, so that validity is the same at every n.

We can easily generalize existing RA constructions to CMRA constructions by lifting the step-indexed validity. For example, for $+_\perp$ from §2.5, this is done as follows:

$$\mathcal{V}_n \triangleq \big\{ \mathsf{inl}(a_1) \mid a_1 \in \mathcal{V}'_n \big\} \cup \big\{ \mathsf{inr}(a_2) \mid a_2 \in \mathcal{V}''_n \big\}$$

3.4 The Model of Iris 2.0

We now have everything set up to generalize Iris to higher-order ghost state. Instead of the user picking a fixed RA R, they can pick an arbitrary CMRA M *that can depend on iProp*. Formally, this dependency is expressed by a *function* $F : \mathcal{COFE} \to \mathcal{CMRA}$:

$$iProp \approx World \xrightarrow{\mathrm{mon}} Res \xrightarrow{\mathrm{mon}} SProp \qquad (\text{IRIS-PRE-2.0})$$

$$\text{where} \quad World \triangleq \mathbb{N} \xrightarrow{\mathrm{fin}} \blacktriangleright iProp$$

$$Res \triangleq F(iProp) \times \Delta\mathrm{Ex}(PhyState)$$

Notice that this is very similar to IRIS-1.1. The only difference is that *Res* is no longer discrete, but rather defined in terms of F. Similar to the case of the $\blacktriangleright$ in *World*, every recursive occurrence of *iProp* in F needs to be guarded by a $\blacktriangleright$ to ensure the existence of a fixed point of the recursive domain equation. (Formally that means F should be *locally contractive* [2, 7].)

In order to solve the challenge outlined in §1.1, recall that we wish to obtain an instantiation of Iris where the ghost state is a heap of oneshots of type T, where T may depend on *iProp*. Let us rewrite it as $T = G(iProp)$ to factor out the dependency on *iProp*. We can then choose the following function for the ghost state:

$$F(X) \triangleq \mathbb{N} \xrightarrow{\mathrm{fin}} \mathrm{ONESHOT}(\mathrm{AG}(\blacktriangleright G(X)))$$

We will see in §4.2 that this suffices to prove the correctness of the client from §1.1.

Since F is a function between types in a category, we require it to be a *bifunctor* from $\mathcal{COFE}$ to $\mathcal{CMRA}$.[6] This is typically easy to show. Using bifunctors, we can handle both co- and contravariant occurrences of F's argument, *e.g.*, if the type T is $iProp \to iProp$, in which case $G(X) \triangleq X \xrightarrow{\mathrm{ne}} X$. Of course, if all you need is first-order ghost state, you can easily pick the constant functor

---

[6] The arrows of $\mathcal{CMRA}$ are monotone, non-expansive functions—for further details, see our technical appendix [1].

$F(\_) = M$ for any CMRA or RA M. In this case, there are no further proof obligations; F trivially satisfies all the functor laws.

So, to sum this up: Iris 2.0 is parameterized over a user-selected locally contractive bifunctor F. Within the logic, the user then has $F(iProp)$ available for the ghost state, which gives rise to arbitrary higher-order ghost state in a canonical and principled way.

Unifying worlds and resources. There is one final change in the model of Iris 2.0, compared to IRIS-1.1. Looking at IRIS-PRE-2.0, we see that *World* and *Res* are recursively referring to *iProp*. By equipping worlds with an appropriate CMRA structure, we can merge worlds into the resources, which means we can make resources comprise physical state, ghost state, *and* worlds.

What we need for this is a CMRA structure on *World*—but it turns out that we have already defined the right CMRA, namely, AG. One important property of invariants is that everybody involved *agrees* on what the invariant with a particular name is; the agreement CMRA ensures that this is the case.

This leads us to the following equation for defining the semantic model of Iris:

$$iProp \approx Res \xrightarrow{\mathrm{mon}} SProp \qquad (\text{IRIS-2.0})$$

$$\text{where} \quad Res \triangleq World \times F(iProp) \times \Delta\mathrm{Ex}(PhyState)$$

$$World \triangleq \mathbb{N} \xrightarrow{\mathrm{fin}} \mathrm{AG}(\blacktriangleright iProp)$$

Ignoring the AG, this is just the uncurried form of IRIS-PRE-2.0. However, as a consequence of this, *iProp* is now just a predicate over a CMRA. Practically speaking, this means that most Iris connectives can be modeled independently of worlds. Almost all connectives can be interpreted in $M \xrightarrow{\mathrm{mon}} SProp$, completely generic in the choice of the CMRA M. This kind of abstraction and unification simplifies proving soundness of the logical rules.

Note that IRIS-2.0 brings us very close to the canonical model of traditional separation logic—using a predicate over PCMs—the core difference being step-indexing. This goes to show that even the most complicated concurrent impredicative higher-order separation logics can be semantically reduced to the same fundamental idea. In that sense, equipping Iris with generic higher-order ghost state actually *simplifies* its model.

4. Case Study: Barrier

In this section, we will show how to implement and verify the example from §1.1 using a synchronization primitive called a *barrier* (which one can also think of as a "oneshot channel"). We follow the implementation and specification of this primitive as presented in Dodds *et al.* [15]. As we will see, higher-order ghost state is crucially useful both in our use of the barrier and in verifying the correctness of the barrier itself.

A barrier is a synchronization primitive offering two actions:

- Clients can `wait` on the barrier, which will block these processes until the barrier is triggered.

- A client can `signal` the barrier, which will cause all waiting clients to resume execution.

You can find the code implementing a barrier in Figure 9. This implementation is fairly straightforward: A waiting client is in a busy loop until it sees the location x updated to 1. So, all that `signal` has to do is set the location to 1, and all current and future waiting clients will leave their busy loop.

The example client in Figure 11 shows how a barrier can be used to implement the pattern described in §1.1. Notice that we are actually using *the same* barrier to communicate with both worker threads, which is possible because a barrier can trigger multiple threads when signaled. In fact, the client given here is even more general than the one sketched in §1.1, since the worker threads

$$\texttt{newbarrier} \triangleq \lambda\_.\ \texttt{ref}(0)$$

$$\texttt{signal} \triangleq \lambda x.\ x \leftarrow 1$$

$$\texttt{wait} \triangleq \texttt{rec}\ wait(x) = \texttt{if}\ !x = 1\ \texttt{then}\ ()\ \texttt{else}\ wait(x)$$

Figure 9. Implementation of a barrier

NEWBARRIER
$$\{\textsf{True}\}\ \texttt{newbarrier}()\ \{\ell.\ \textsf{send}(\ell, P) * \textsf{recv}(\ell, P)\}$$

SIGNAL
$$\{\textsf{send}(\ell, P) * P\}\ \texttt{signal}(\ell)\ \{\textsf{True}\}$$

WAIT
$$\{\textsf{recv}(\ell, P)\}\ \texttt{wait}(\ell)\ \{P\}$$

$$\frac{\text{RECV-MONO} \quad P \vdash Q}{\textsf{recv}(\ell, P) \vdash \textsf{recv}(\ell, Q)}$$

RECV-SPLIT
$$\textsf{recv}(\ell, P_1 * P_2) \Rrightarrow \textsf{recv}(\ell, P_1) * \textsf{recv}(\ell, P_2)$$

Figure 10. Barrier specification

are *transforming* their P_i into Q_i, and the main thread later joins these two together to obtain Q. Before we go into details about the verification of the client, let us have a closer look at specifying the interface provided by the barrier library.

4.1 Barrier Specification

The behavior of the barrier is specified in Figure 10. When initializing a barrier using $\texttt{newbarrier}()$, the resources $\textsf{recv}(\ell, P)$ and $\textsf{send}(\ell, P)$ are created, which can then be used by the waiting thread and signaling thread, respectively. The assertion P describes the resources the barrier has to transfer, so when calling $\texttt{signal}(\ell)$, the resource P has to be given up and is transferred to the waiting client. Conversely, when $\texttt{wait}(\ell)$ returns, the waiting client receives P.

Notice that $\texttt{newbarrier}$ works for *any* assertion P, without the caller having to establish P or give up any resources on initialization of the barrier. In the example shown in Figure 11, the assertion P is in fact only established by the "expensive computation", long after the barrier has been created.

The predicates $\textsf{recv}(\ell, P)$ and $\textsf{send}(\ell, P)$ are *impredicative higher-order abstract predicates*. This means that they are predicates describing any assertion P (higher-order), which can include occurrences of $\textsf{recv}$ and $\textsf{send}$ itself (impredicative), as well as *e.g.,* Hoare triples. It is thus for example possible to send the permission for code pointers through the barrier. Finally, these predicates are defined by the implementation of the barrier, but their details are not exposed to the client (abstract).

The rule RECV-SPLIT plays an important role in proving the correctness of the client in Figure 11. The client has *two* threads that both call $\texttt{wait}$, and thus both need a $\textsf{recv}$ resource, whereas after initialization of the barrier we have just one such resource. If we can *split* P into two pieces P_1 and P_2, then we can use RECV-SPLIT to split $\textsf{recv}(\ell, P)$ into $\textsf{recv}(\ell, P_1)$ and $\textsf{recv}(\ell, P_2)$. Splitting makes it possible to move the resources $\textsf{recv}(\ell, P_1)$ and $\textsf{recv}(\ell, P_2)$ to different threads, which then can both call $\texttt{wait}$ to obtain their piece of the resource transmitted through the barrier.

4.2 Splitting and Joining the Existentials

For the verification of the example client in Figure 11, we assume that we have the following Hoare triples for the main thread M and the workers W_i for $i \in \{1, 2\}$:

$$\{P'\}\ e\ \{\exists x{:}T.\ P\} \qquad \forall x{:}T.\ \{P_i\}\ e_i\ \{Q_i\}$$

Here and below, the assertions P, P_i, Q_i and Q range over a variable x of type $T = G(iProp)$ for some bifunctor G (*cf.* §3.4). We furthermore assume that resources can be split and merged.

$$\forall x{:}T.\ P \Rightarrow P_1 * P_2 \qquad \forall x{:}T.\ Q_1 * Q_2 \Rightarrow Q$$

Oneshot ghost variables. The functor that we need to implement oneshot ghost variables is $F(X) \triangleq \textsc{OneShot}(\textsc{Ag}(\blacktriangleright G(X)))$, so the ghost state is of type $F(iProp) = \textsc{OneShot}(\textsc{Ag}(\blacktriangleright T))$. We encode oneshot ghost locations as follows:

$$\gamma \hookrightarrow \textsf{pending} \triangleq \boxed{\textsf{inl}()}^\gamma$$

$$\gamma \hookrightarrow x \triangleq \boxed{\textsf{inr}(\textsf{ag}(\textsf{next}(x)))}^\gamma \qquad \text{for } x : T$$

Notice that $\gamma \hookrightarrow x$ is a persistent assertion stating that the oneshot location has been initialized. The state of location γ is obtained by injecting x into $\textsc{OneShot}(\textsc{Ag}(\blacktriangleright T))$.

From generic properties of the oneshot and agreement CMRA we obtain the following proof rules to initialize a previously uninitialized location, and to encode that a location, once initialized, will not change its value:

$$\gamma \hookrightarrow \textsf{pending} \Rrightarrow \gamma \hookrightarrow x \qquad (\textsc{oneshot-init})$$

$$\gamma \hookrightarrow x * \gamma \hookrightarrow y \Rightarrow \rhd(x = y) \qquad (\textsc{oneshot-agree-later})$$

We will now prove the latter rule (ONESHOT-AGREE-LATER) to show that it is indeed a derived rule. First of all, observe that owning $\gamma \hookrightarrow x * \gamma \hookrightarrow y$ is logically equivalent to (using GHOST-OP):

$$\boxed{\textsf{inr}(\textsf{ag}(\textsf{next}(x)) \cdot \textsf{ag}(\textsf{next}(y)))}^\gamma$$

By GHOST-VALID we know that we can only own *valid* ghost CMRA elements. In combination with AG-VALID, we thus obtain

$$\textsf{next}(x) = \textsf{next}(y)$$

Finally, we use the following general property of the later construction $\blacktriangleright T$ to complete the proof of ONESHOT-AGREE-LATER:

$$\textsf{next}(x) = \textsf{next}(y) \Leftrightarrow \rhd(x = y)$$

Remember that $\rhd(x = y)$ means that x and y are equal at the *next* step-index. This is not a problem, since we can get rid of the $\rhd$ when the program performs a physical step of execution.

Verification of the worker threads. We decorated Figure 11 with verification conditions to document resources available at any particular point in the code. To obtain the resources needed for the worker threads, we want to *split* the $\textsf{recv}$ using RECV-SPLIT, which first requires us to weaken the barrier assertion with RECV-MONO:

$$(\exists x{:}T.\ \gamma \hookrightarrow x * P) \Rightarrow (\exists x{:}T.\ \gamma \hookrightarrow x * P_1) * (\exists x{:}T.\ \gamma \hookrightarrow x * P_2)$$

It is straightforward to verify correctness of the worker threads.

Joining the existentials. What is left to justify is the last line of Figure 11: How can the two existentials be joined together? The core lemma for this step of the proof is ONESHOT-AGREE-LATER. From there, we easily obtain:

$$(\exists x{:}T.\ \gamma \hookrightarrow x * Q_1) * (\exists x{:}T.\ \gamma \hookrightarrow x * Q_1) \Rightarrow \rhd(\exists x{:}T.\ \gamma \hookrightarrow x * Q)$$

The $\rhd$ on the right hand side is then consumed by the parallel composition, when it performs a physical step of computation to join the threads back together.

This completes the verification: We demonstrated that, using the higher-order ghost state as defined in §3, we can solve the challenge posed in §1.1 for existential quantification over *arbitrary types*.

4.3 Verifying the Barrier Specification: Saved Propositions

As has been discussed in [15], it is very difficult to prove a specification for a barrier that includes the splitting axiom. Intuitively, the reason for this is as follows: When a thread calls $\textsf{send}$ to give up
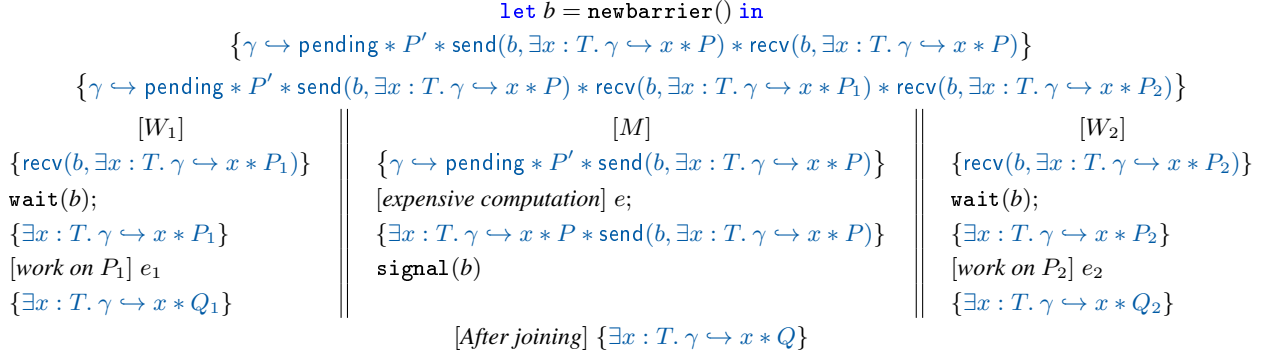
$$\texttt{let } b = \texttt{newbarrier()} \texttt{ in}$$
$$\{\gamma \hookrightarrow \texttt{pending} * P' * \texttt{send}(b, \exists x : T.\ \gamma \hookrightarrow x * P) * \texttt{recv}(b, \exists x : T.\ \gamma \hookrightarrow x * P)\}$$
$$\{\gamma \hookrightarrow \texttt{pending} * P' * \texttt{send}(b, \exists x : T.\ \gamma \hookrightarrow x * P) * \texttt{recv}(b, \exists x : T.\ \gamma \hookrightarrow x * P_1) * \texttt{recv}(b, \exists x : T.\ \gamma \hookrightarrow x * P_2)\}$$

$[W_1]$	$[M]$	$[W_2]$
$\{\texttt{recv}(b, \exists x : T.\ \gamma \hookrightarrow x * P_1)\}$	$\{\gamma \hookrightarrow \texttt{pending} * P' * \texttt{send}(b, \exists x : T.\ \gamma \hookrightarrow x * P)\}$	$\{\texttt{recv}(b, \exists x : T.\ \gamma \hookrightarrow x * P_2)\}$
$\texttt{wait}(b);$	$[\textit{expensive computation}]\ e;$	$\texttt{wait}(b);$
$\{\exists x : T.\ \gamma \hookrightarrow x * P_1\}$	$\{\exists x : T.\ \gamma \hookrightarrow x * P * \texttt{send}(b, \exists x : T.\ \gamma \hookrightarrow x * P)\}$	$\{\exists x : T.\ \gamma \hookrightarrow x * P_2\}$
$[\textit{work on } P_1]\ e_1$	$\texttt{signal}(b)$	$[\textit{work on } P_2]\ e_2$
$\{\exists x : T.\ \gamma \hookrightarrow x * Q_1\}$		$\{\exists x : T.\ \gamma \hookrightarrow x * Q_2\}$

$$[\textit{After joining}]\ \{\exists x : T.\ \gamma \hookrightarrow x * Q\}$$

Figure 11. Example client of a barrier

---

resources P, it has to prove that P matches the separating conjunction of the resources that all waiting clients are expecting to receive. But how should $\texttt{signal}$ know how the resources should be split among the waiting clients? The splitting may have happened after the call to $\texttt{newbarrier}$, and yet it must be somehow communicated to $\texttt{signal}$.

To our knowledge, the only way to do this is to record the desired splitting in the global *ghost state*, which is accessible to both the thread that performs the splitting and to $\texttt{signal}$. This immediately means that we need to be able to have ghost state that depends on propositions—that is, arbitrary propositions of the logic (*iProp*), not just meta-level propositions (*Prop*).

One convenient way of actually going about this proof is to apply *saved propositions* [15]. In Iris, the interface of saved propositions is as follows:

$$\forall P.\ \textsf{True} \Rrightarrow \exists \gamma.\ \gamma \Mapsto P \qquad \text{(SPROP-ALLOC)}$$
$$\forall \gamma, P, Q.\ (\gamma \Mapsto P * \gamma \Mapsto Q) \Rightarrow \triangleright(P \Leftrightarrow Q) \qquad \text{(SPROP-AGREE)}$$

The first rule says that we can allocate a new saved proposition P at a fresh ghost location γ. Notice that to establish a saved proposition, we do not actually have to establish that assertion. The allocated assertion $\gamma \Mapsto P$ is persistent. The second rule then allows us to derive, given two assertions about the *same* location, that they must be about the same proposition.

It is not coincidental that these rules look very similar to GHOST-ALLOC and ONESHOT-AGREE: the oneshot locations described in §4.2 are strictly more general than saved propositions. However, it is easy enough to directly encode saved propositions by defining $F(X) \triangleq \textsc{AG}(\blacktriangleright X)$ to obtain ghost state of the form $\textsc{AG}(\blacktriangleright iProp)$.

The full proof of the barrier specification in Iris is carried out in our technical appendix [1] and is formalized in Coq.

5. Coq Formalization

Adequacy and soundness for Iris 2.0, many derived constructions, and all specifications and proofs in this paper have been fully formalized using Coq. We discuss important aspects of the formalization, features of Coq that were essential, and the key differences between the formalization of Iris 1.0 and Iris 2.0.

5.1 Algebraic Structures

The main challenge of formalizing the Iris logic is that it lives in the category of COFEs instead of the conventional category of Coq $\texttt{Types}$. So, instead of using plain Coq $\texttt{Types}$, we use types equipped with an ordered family of equivalences, and instead of plain Coq functions, we use non-expansive functions. This poses some interesting challenges:

- In Iris we use large nested COFE constructions composed from smaller COFEs (see for example the construction IRIS-2.0). Coq should be able infer the COFE structure on these nested constructions automatically without considerable overhead.

- Defining non-expansive functions should be lightweight. Most importantly, just to *write down* a $\forall x{:}T.\ P\,x$ or $\exists x{:}T.\ P\,x$ quantification in the logic (where $P : T \xrightarrow{\text{ne}} iProp$), there should be no proof obligations.

- Rewriting with the equivalence relations $\stackrel{n}{=}$ of COFEs should work fast and reliably. Note that rewriting with $\stackrel{n}{=}$ is only possible in the context of non-expansive functions, and thus requires Coq to derive facts about non-expansiveness. In Coq, rewriting with user-defined equivalence relations is called *setoid* rewriting.

The Coq formalization of Iris 1.0, which used the ModuRes library [32], did not meet any of these challenges. Dealing with large COFE constructions made Coq very slow, setoid rewriting worked unreliably, and proofs of non-expansiveness cluttered definitions. The Coq formalization of Iris 2.0 has been implemented from scratch, and we have improved on all of these points.

In order to emphasize the difficulties involved, let us describe the two most commonly used patterns of representing algebraic structure (like COFEs and CMRAs in our case) in Coq.

- The *bundled approach* consists of representing algebraic structures as dependently typed records containing the carrier, the operations, and proofs. COFEs would thus be formalized as:

```
Record cofeT := {
  cofe_car :> Type;
  cofe_dist : nat → relation cofe_car;
  cofe_dist_equivalence : ∀ n,
    Equivalence (cofe_dist n);
  (* ... *)
}.
```

- The *unbundled approach* consists of representing algebraic structures as predicates over the carrier, the operations, and proofs. COFEs would thus be formalized as:

```
Record cofeT {A} (d : nat → relation A) := {
  cofe_dist_equivalence : ∀ n,
    Equivalence (cofe_dist n);
  (* ... *)
}.
```

In order to automatically infer algebraic structures given the `Type` of the carrier, Coq supports two similar mechanisms: canonical structures and type classes. Canonical structure search is guided by the *fields* of a record, and is thus commonly used in combination with the bundled approach [19]. Type class search, on the other hand, is guided by the *parameters* of a record, and is thus commonly used in combination with the unbundled approach [34].

Iris 1.0 used the unbundled approach to represent COFEs, which is known to suffer from an exponential blow-up in term size when nesting instances of algebraic structures. In order to remedy this shortcoming, Iris 2.0 uses a bundled approach using canonical structures inspired by Ssreflect's [19].

Iris 1.0 demanded proofs of non-expansiveness of functions *a priori* pretty much everywhere. We changed this in Iris 2.0 such that for example the predicates in universal and existential quantification, and the post-condition of Hoare triples, do not *a priori* have to be non-expansive. This allows us to reuse Coq's functions and notation when writing down formulas in the Iris logic. Proofs of non-expansiveness are only needed for setoid-rewriting and constructing recursive definitions, but not for validity of the proof rules, and can thus be established *a posteriori*. This approach matches well with Coq's setoid machinery, which uses type classes to register compatibility with equivalence relations *a posteriori* [33].

Although Iris 2.0 made considerable progress on the formalization challenges we posed, we do not consider these challenges to be solved. Since we need the combination of bundled algebraic structures and setoid rewriting, we are using a mixture of canonical structures and type classes, whose interaction is somewhat fragile. Most notably, many Coq tactics, including the `rewrite` tactic and the type class machinery, are very sensitive to terms that are only equal up to conversion (*e.g.,* unfolding of canonical structure projections), and they often fail to infer type class arguments. In order to work around these problems, we are using the Ssreflect `rewrite` tactic, which deals better with these issues.

We thus welcome some machinery that unifies type classes and canonical structures in a streamlined way. Unification hints may be a step in this direction [5].

5.2 Programming Language

Similar to Iris 1.0, the Coq formalization of Iris 2.0 is parameterized by the programming language that one wishes to reason about. For the purpose of this paper, we have instantiated Iris with a deeply embedded untyped ML-like language with higher-order store and the concurrency primitives `fork` and `CAS` (compare-and-swap). This improves on the Iris 1.0 formalization, which has not been instantiated with an actual programming language.

The main goal of the Iris 2.0 Coq formalization is verification of actual code, and as such, readability of programs written in the deeply embedded language becomes important. In particular:

1. We want to be able to write programs in a syntax that resembles ML, and we want programs to be pretty printed similarly to how they appear when being interactively verified.

2. Variable binding should be human-readable, hence a "machine oriented" approach like De Bruijn indexes does not suffice.

3. All programs should be *closed*, *i.e.,* they should not contain free variables. The language formalization should ensure closedness of programs automatically.

We make use of Coq's expressive notation mechanism to obtain an ML-like notation for our language, we make use of named variables to make binding human-readable, and we make use of dependent types to ensure closedness of programs. Our representation of programming language expressions is roughly as follows:

```
Class VarBound (x : string) (X : list string) :=
  var_bound : if decide (x∈X) then True else False.
Inductive expr (X : list string) : Type :=
  | Var x : VarBound x X → expr X
  | App    : expr X → expr X → expr X
  | Lam x : expr (x :: X) → expr X.
Coercion App : expr >-> Funclass.
Notation "' x" := (Var x _).
Notation "λ: x..y , e" := (Lam x..(Lam y e)..).
```

The type `expr X` represents expressions whose free variables are a subset of `X`, and as such, expressions of type `expr []` are guaranteed to be closed. An important part of this definition is that the condition x ∈ X for variables is decidable, and can thus be inferred by computation while type checking concrete programs. This is achieved by instrumenting the type class mechanism with a hint to solve `VarBound` constraints by computation:

```
Hint Extern 0 (VarBound _ _) =>
  vm_compute; exact I : typeclass_instances.
```

Notice that we can use a named approach to variable binding without having to worry about variable capture because our language has weak reduction—*i.e.,* we do not reduce under λs.

Example. The Coq version of the code of the barrier implementation is as follows:[7]

```
Definition newbarrier : val := λ: <>, ref #0.
Definition signal : val := λ: "x", '"x" <- #1.
Definition wait : val :=
  rec: "wait" "x" :=
  if: !'"x" = #1 then #() else '"wait" '"x".
```

5.3 Notable Formalization Aspects

Weakest precondition. The Hoare triples of Iris are defined in terms of a more primitive *weakest precondition* judgment. Since the weakest preconditions are better suited for interactive proving, we use them in our Coq proofs. Hoare-triple-based specifications—such as those in §2 and §4.1—are only established at the end.

Non-expansive functions. We have implemented a tactic to prove that functions are non-expansive by repeatedly applying congruence properties. Contrary to Coq's `solve_proper`, our tactic is more efficient and handles `match` constructs.

Combining functors. A core feature of Iris is that the structure of the ghost state can be chosen by the user. This is achieved by parameterizing the logic by a functor F (§3.4). However, when verifying programs that consist of multiple program modules, it is likely that different program modules are in need of different functors G_i. We have used Coq's type class machinery to implement infrastructure that automatically combines these G_i to form the global functor F, so the modules can be combined in one proof.

5.4 Overview

The Coq development, which is entirely constructive and axiom-free, consists of the parts described in Figure 12. Line counts exclude whitespace and comments. We used a support library by Krebbers [25] that contains many definitions and theorems about data structures such as lists, finite sets, and finite maps.

The compilation times of Iris 2.0 have improved by an order of magnitude: Compilation of the components 'Algebra' and 'Program logic' takes less than 2 minutes for Iris 2.0 vs. 23 minutes for Iris 1.0 (using an Intel Core i5-2450M, 4 threads, 2 cores).

---

[7] In this code, `<>` denotes an anonymous binder.

Component	What else	LOC
Algebra	Solver for domain equations	5.444
Program logic	Model, adequacy, derived notions	2.385
Heap language	Derived rules, automation	1.591
Barrier	Higher-order client (see [1])	433
Examples	Examples from §2 and §4.2	239
Total		10.092

Figure 12. Overview of the Coq development

6. Related Work

User-defined higher-order ghost state. Several prior logics support custom (user-defined) ghost state, via Concurroids [27] or PCMs [24, 39]. 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.

Some logics do have a model that can be considered to involve *second-order* ghost state, *e.g.,* to justify their use of dynamically allocated locks [20, 21, 8], regions [35], or invariants [24]. However, this ghost state has a fixed structure, rather than being an instance of a generic algebraic structure like CMRAs.

In some of these prior logics, 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.* [15] showed that the built-in "protocol" mechanism of the iCAP logic [35] can be repurposed to encode *saved propositions* (§4.3), 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 (such as we relied on in proving the example from §1.1).

The Verified Software Toolchain (VST) of Appel *et al.* [4] provides a general framework for defining higher-order logics via "indirection theory" [22]. 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 CMRAs. However, 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 CMRAs can be safely composed (§2.6), whereas proofs carried out in different VST logics do not necessarily interoperate. Furthermore, VST has so far only been applied to sequential and coarse-grained concurrent code, not to fine-grained concurrent algorithms.

PCMs with a (duplicable) core. There have been several presentations of variants of PCMs that have "multiple units" or include a notion of a "duplicable core".

Dockins *et al.* introduced *multi-unit separation algebras* [14], 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 crucial difference between multi-unit separation algebras and RAs is that we present the monoidal operation and core as a function, whereas they represent these as relations.

The terminology of a *(duplicable) core* has been adapted from Pottier, who introduced it in the context of *monotonic separation algebras* [30]. However, the axioms of Pottier's cores, as well as those of related notions in other work [39, 4], are somewhat different

from the axioms of our RAs. Some common properties appear consistently (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),[8] 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 §2.5, partial cores make it easier to compose RAs out of simpler constructions like sums.

Coq formalizations. Over the past decade, there has been tremendous progress on formalization of program logics in proof assistants, with focus on supporting strong proof automation, as well as the ability to deal with realistic programming languages (see, *e.g.,* [4, 31, 6, 23, 9, 25, 39, 37]).

However, with the exception of the aforementioned VST, these formalized program logics do not support impredicative invariants (§2.2), let alone higher-order ghost state.

7. Conclusion

We have introduced higher-order ghost state and have shown that it is a useful extension of higher-order concurrent separation logic. Moreover, we have presented Iris 2.0, an extension of the original Iris program logic with support for higher-order ghost state.

Soundness of Iris 2.0 is proven using a new model construction, which, in addition to supporting higher-order ghost state, also unifies two of the core model concepts, worlds and resources, that were distinct before. This simplifies the meta-theory, which is important not only conceptually, but also when formalizing the meta-theory of Iris in a proof assistant. We have formalized Iris 2.0 in Coq, and the formalization itself is a strong improvement over the earlier formalization of Iris 1.0.

In ongoing work, we are using the new Iris 2.0 formalization to conduct larger experiments with Iris. To support such experiments, we are in the process of developing tactic support for reasoning about concurrent higher-order programs.

In order to facilitate interactive reasoning in Iris using Coq, we have extended Coq with support for the linear context of separation logic. This extension provides named assumptions and tactical support for introduction and case analysis of these linear assumptions. Initial experiments are promising and show a significant improvement in ergonomics compared to using the rules of Iris manually.

In addition, we plan to develop tactics to reason fully automatically about client code. This will probably be done using Malecha and Bengtson's recent reflective tactics [26].

---

[8] The RA of STSs, as well as the example demonstrating that the core is not a homomorphism, is described in our technical appendix [1].

References

[1] Higher-Order Ghost State: Appendix and Coq development. Available on the Iris project website at http://plv.mpi-sws.org/iris/.

[2] P. America and J. Rutten. Solving reflexive domain equations in a category of complete metric spaces. *JCSS*, 39(3):343–375, 1989.

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

[4] A. W. Appel, editor. *Program Logics for Certified Compilers*. Cambridge University Press, 2014.

[5] A. Asperti, W. Ricciotti, C. S. Coen, and E. Tassi. Hints in unification. In *TPHOLs*, volume 5674 of *LNCS*, pages 84–98, 2009.

[6] J. Bengtson, J. B. Jensen, and L. Birkedal. Charge! - A Framework for Higher-Order Separation Logic in Coq. In *ITP*, volume 7406 of *LNCS*, pages 315–331, 2012.

[7] L. Birkedal, K. Støvring, and J. Thamsborg. The category-theoretic solution of recursive metric-space equations. *TCS*, 411(47):4102–4122, 2010.

[8] A. Buisse, L. Birkedal, and K. Støvring. Step-indexed Kripke model of separation logic for storable locks. *ENTCS*, 276:121–143, 2011.

[9] A. Chlipala. The Bedrock structured programming system: combining generative metaprogramming and Hoare logic in an extensible program verifier. In *ICFP*, pages 391–402, 2013.

[10] E. Cohen, E. Alkassar, V. Boyarinov, M. Dahlweid, U. Degenbaev, M. Hillebrand, B. Langenstein, D. Leinenbach, M. Moskal, S. Obua, W. Paul, H. Pentchev, E. Petrova, T. Santen, N. Schirmer, S. Schmaltz, W. Schulte, A. Shadrin, S. Tobies, A. Tsyban, and S. Tverdyshev. Invariants, modularity, and rights. In *PSI*, volume 5947 of *LNCS*, pages 43–55, 2009.

[11] P. da Rocha Pinto, T. Dinsdale-Young, and P. Gardner. TaDA: A logic for time and data abstraction. In *ECOOP*, pages 207–231, 2014.

[12] T. Dinsdale-Young, L. Birkedal, P. Gardner, M. J. Parkinson, and H. Yang. Views: Compositional reasoning for concurrent programs. In *POPL*, 2013.

[13] T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP*, pages 504–528, 2010.

[14] R. Dockins, A. Hobor, and A. W. Appel. A fresh look at separation algebras and share accounting. In *APLAS*, pages 161–177, 2009.

[15] M. Dodds, S. Jagannathan, M. J. Parkinson, K. Svendsen, and L. Birkedal. Verifying custom synchronization constructs using higher-order separation logic. *TOPLAS*, 38(2):4, 2016.

[16] X. Feng. Local rely-guarantee reasoning. In *POPL*, pages 315–327, 2009.

[17] X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP*, pages 173–188, 2007.

[18] M. Fu, Y. Li, X. Feng, Z. Shao, and Y. Zhang. Reasoning about optimistic concurrency using a program logic for history. In *CONCUR*, pages 388–402, 2010.

[19] F. Garillot, G. Gonthier, A. Mahboubi, and L. Rideau. Packaging mathematical structures. In *TPHOLs*, volume 5674 of *LNCS*, pages 327–342, 2009.

[20] A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local reasoning for storable locks and threads. In *APLAS*, pages 19–37, 2007.

[21] A. Hobor, A. Appel, and F. Zappa Nardelli. Oracle semantics for concurrent separation logic. In *ESOP*, pages 353–367, 2008.

[22] A. Hobor, R. Dockins, and A. Appel. A theory of indirection via approximation. In *POPL*, 2010.

[23] J. B. Jensen, N. Benton, and A. Kennedy. High-level separation logic for low-level code. In *POPL*, pages 301–314, 2013.

[24] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*, pages 637–650, 2015.

[25] R. Krebbers. *The C standard formalized in Coq*. PhD thesis, Radboud University, 2015.

[26] G. Malecha and J. Bengtson. Easy and efficient automation through reflective tactics. In *ESOP*, 2016.

[27] A. Nanevski, R. Ley-Wild, I. Sergey, and G. A. Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *ESOP*, pages 290–310, 2014.

[28] P. O'Hearn. Resources, concurrency, and local reasoning. *TCS*, 375(1):271–307, 2007.

[29] S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *CACM*, 19(5):279–285, 1976.

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

[31] I. Sergey, A. Nanevski, and A. Banerjee. Mechanized verification of fine-grained concurrent programs. In *PLDI*, pages 77–87, 2015.

[32] F. Sieczkowski, A. Bizjak, and L. Birkedal. ModuRes: A Coq library for modular reasoning about concurrent higher-order imperative programming languages. In *ITP*, volume 9236 of *LNCS*, pages 375–390, 2015.

[33] M. Sozeau. A new look at generalized rewriting in type theory. *JFR*, 2(1):41–62, 2009.

[34] B. Spitters and E. van der Weegen. Type classes for mathematics in type theory. *MSCS*, 21(4):795–825, 2011.

[35] K. Svendsen and L. Birkedal. Impredicative concurrent abstract predicates. In *ESOP*, pages 149–168, 2014.

[36] K. Svendsen, L. Birkedal, and M. J. Parkinson. Modular reasoning about separation of concurrent data structures. In *ESOP*, pages 169–188, 2013.

[37] H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In *POPL*, pages 97–108, 2007.

[38] A. Turon, D. Dreyer, and L. Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*, pages 377–390, 2013.

[39] A. Turon, V. Vafeiadis, and D. Dreyer. GPS: navigating weak memory with ghosts, protocols, and separation. In *OOPSLA*, pages 691–707, 2014.

[40] V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, pages 256–271, 2007.