Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning

Ralf Jung
MPI-SWS &
Saarland University
jung@mpi-sws.org

David Swasey
MPI-SWS
swasey@mpi-sws.org

Filip Sieczkowski
Aarhus University
filips@cs.au.dk

Kasper Svendsen
Aarhus University
ksvendsen@cs.au.dk

Aaron Turon
Mozilla Research
aturon@mozilla.com

Lars Birkedal
Aarhus University
birkedal@cs.au.dk

Derek Dreyer
MPI-SWS
dreyer@mpi-sws.org

Abstract

We present Iris, a concurrent separation logic with a simple premise: *monoids and invariants are all you need*. Partial commutative monoids enable us to express—and invariants enable us to enforce—user-defined *protocols* on shared state, which are at the conceptual core of most recent program logics for concurrency. Furthermore, through a novel extension of the concept of a *view shift*, Iris supports the encoding of *logically atomic specifications*, *i.e.,* Hoare-style specs that permit the client of an operation to treat the operation essentially as if it were atomic, even if it is not.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.3 [*Programming Languages*]: Language Constructs and Features; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords Separation logic, fine-grained concurrency, atomicity, partial commutative monoids, invariants, higher-order logic, compositional verification.

1. Introduction

Concurrency is fundamentally about shared state. This is true not only for shared-memory concurrency, where the state takes the form of a "heap" that threads may write to and read from, but also for message-passing concurrency, where the state takes the form of a "network" that threads may send to and receive from (or a sequence of "events" on which threads may synchronize). Thus, to scalably verify concurrent programs of any stripe, we need compositional methods for reasoning about shared state.

This goal has sparked a long line of work, especially in recent years, during which a synthesis of *rely-guarantee reasoning* [21] and *separation logic* [31, 28] has led to a series of increasingly advanced program logics for concurrency: RGSep [37], SAGL [13], LRG [12], CAP [10], HLRG [15], CaReSL [34], iCAP [33], FCSL [27], TaDA [8], and others. In this paper, we present a logic called **Iris** that explains some of the complexities of these prior separation logics in terms of a simpler unifying foundation, while also supporting some new and powerful reasoning principles for concurrency.

Before we get to Iris, however, let us begin with a brief overview of some key problems that arise in reasoning compositionally about shared state, and how prior approaches have dealt with them.

1.1 Invariants and their limitations

The canonical model of concurrency is *sequential consistency* [23]: threads take turns interacting with the shared state (reading/writing, sending/receiving), with each turn lasting for one step of computation.[1] Although the semantics of sequentially consistent (SC) concurrency is simple to define, that does not mean it is easy to reason about. In particular, the key question is how to do *thread-local* reasoning—that is, verifying one thread at a time—even though other threads may *interfere* with (*i.e.,* mutate) the shared state in between each step of computation in the thread we are verifying.

The invariant rule. The simplest (and oldest) way in which concurrent program logics account for such interference is via *invariants* [5]. An invariant is a property that holds of some piece of shared state at all times: each thread accessing the state may assume the invariant holds *before* each step of its computation, but it must also ensure that it continues to hold *after* each step.

Formally, in concurrent separation logics, the invariant rule looks *something like* the following (omitting some important details that we explain later in §4):

$$\frac{\{R * P\}\ e\ \{R * Q\} \qquad e \text{ physically atomic}}{\boxed{R} \vdash \{P\}\ e\ \{Q\}}$$

Here, the assertion $\boxed{R}$ states the knowledge that there exists an invariant R governing some piece of shared state. Given this knowledge, the rule tells us that e may gain (exclusive) control of the shared state satisfying R, so long as it ensures that R continues to hold of it when it is finished executing. Note the crucial side condition that e be *physically atomic*, meaning that it takes exactly one step of computation. If e were not physically atomic, then another thread might access the shared state governed by $\boxed{R}$ during e's execution, in which case it would not be safe for the rule to grant e exclusive control of the shared state throughout its execution.

[1] There is much recent work on weaker models of concurrency, which are in many ways more realistic, but in this paper we focus on SC concurrency.

POPL '15, January 15–17, 2015, Mumbai, India.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3300-9/15/01...$15.00.
http://dx.doi.org/10.1145/2676726.2676980

The invariant rule is simple and elegant. Unfortunately, it also suffers from two major limitations, which a significant amount of follow-on work has attempted to overcome.

Limitation #1: The need for protocols. The first limitation pertains to the seemingly static nature of invariants. For the kinds of interference found in more sophisticated concurrent programs, one may require something a bit more "dynamic" than fixed invariants. It is often necessary, for instance, to have a way of expressing:

- *Irreversibility:* once a certain change to some shared state has occurred, it is irreversible—we cannot go back.
- *Rights:* certain changes to the shared state may only be made by privileged threads that have the "rights" to make those changes.

At first glance, at least, neither of these seems to be expressible in the limited language of invariants. As a consequence, many more recent logics provide some mechanism—such as rely-guarantee assertions [21, 37, 12], regions [10, 33, 8], STSs [34], and concurroids [27]—to account for irreversibility and rights. All of these mechanisms are effectively different ways of describing *protocols* on shared state, which assert how the shared state is permitted to evolve over time.

Although the protocol mechanisms of modern logics are clearly useful, each logic bakes in its own somewhat different mechanism (and corresponding proof rules) as primitive, leading us to wonder: Is this really necessary? *Might there be a simpler logical mechanism for legislating interference, from which more advanced mechanisms could be easily derived?*

Limitation #2: The need for logical atomicity. The second limitation pertains to the side condition of the invariant rule requiring the operation e to be physically atomic (take exactly one step of computation). Obviously, this side condition makes it much more desirable (as a client) to program with physically atomic operations than with non-atomic ones. But there are many operations that *appear* to be atomic even though they take more than one step of computation, and for such operations the invariant rule is not applicable.

Specifically, it is often desirable to construct concurrent programs so that the interference on shared state is only observable within the limited scope of some mutable ADT (abstract data type). For instance, consider concurrent stacks. A sophisticated implementation [16] may rely on fine-grained synchronization between threads in order to maximize parallelism, and thus proving it correct demands the use of invariants (or protocols) to account for the rampant interference. But this internal interference need not infect the verification of client code. In particular, the canonical notion of correctness for concurrent stacks is that it should appear to clients as if all stack operations take effect atomically (*i.e.,* in some sequential order), which is typically formalized via *linearizability* [17] or *contextual refinement* [14, 34]. In this case, we say that the operations provided by the stack ADT are *observably atomic*.

Ideally, we would like to be able to treat observably atomic operations *as if they were* physically atomic. That way, clients of the stack ADT could establish invariants on the contents of the stack and then use the invariant rule when reasoning about push and pop operations. This *can* be done to some extent with contextual refinement (see the layered verification in [34]), but it requires going *outside the logic* in order to connect up the verification of an ADT with that of its clients. Moreover, the refinement approach is only applicable in higher-level languages where the type system is strong enough to support hiding the internals of an ADT from its clients.

A more flexible approach, we argue, would be to *internalize* the notion of observable atomicity as a logical specification, so that both the proof obligation for establishing atomicity and the proof principle it provides to clients take the form of a single Hoare-style specification written *within the logic itself*. We refer to such a

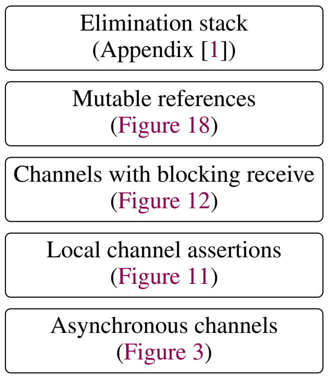

Figure 1. Case study: A stack of abstractions.

specification as being *logically atomic*. Intuitively, the key benefit of working with a logically atomic spec is to enable a version of the invariant rule *without the side condition that e be physically atomic*. This allows clients to treat operations providing such specs as if they were actually atomic, without requiring any extra-logical reasoning.

Very recent work by da Rocha Pinto *et al.* on their logic TaDA [8] supports precisely such a notion of logically atomic specs, together with a number of useful but nonstandard proof rules, but all of these are baked into the logic as primitive. Furthermore, TaDA is currently not able to reason about a significant class of sophisticated concurrent data structures that employ inter-thread cooperation (aka "helping") [35, 33].

This begs the question: *Might there be a way to define logical atomicity in terms of simpler logical primitives, and to derive the associated reasoning principles within the logic?*

1.2 Iris: An orthogonal basis for concurrent reasoning

In this paper, we propose Iris, a concurrent separation logic based on a very simple premise: *monoids and invariants are all you need*.

Invariants we have already discussed at length. As for monoids: Partial commutative monoids (PCMs) are widely known to provide a generic model of *resources*—both physical resources (like the heap) and logical resources (like ghost state)—suitable for use in a separation logic. A number of modern logics involve PCMs either in the model of the logic [33, 34] or as a feature in the logic itself [22, 24, 9]. What does not seem to be widely known, however, is that monoids and invariants form a kind of orthogonal basis for concurrent reasoning. In particular:

- Monoids enable us to *express* protocols on shared state.
- Invariants enable us to *enforce* protocols on shared state.

These two mechanisms, put together, are unexpectedly powerful. In particular, using monoids and invariants, we show how to:

- Encode a variety of the advanced *protocol* mechanisms from modern logics (§2 and §3).
- Derive *proof rules* that are taken as primitive proof rules in other modern logics (§4 and §5).
- Encode TaDA-style *logically atomic specs*, and derive their most important associated proof rules (§7).
- Go beyond TaDA and verify logically atomic specs for fine-grained concurrent ADTs that employ *helping* (§8).

A key technical novelty of Iris that facilitates the above contributions is an extension of the concept of *view shift* that has been proposed in recent work [10, 9, 33]. We explain this mechanism in §4. Furthermore, the soundness of Iris's primitive proof rules, which we discuss briefly in §6, has been fully mechanized in Coq [1].

Syntax

(Fix disjoint, infinite sets *Chan* and *Var* of channel names and variables.)

$$c, d \in \mathit{Chan}$$
$$x, y, f \in \mathit{Var}$$
$$v, w ::= x \mid c \mid \textsf{rec}\ f(x).\ e \mid () \mid (v, v) \mid \textsf{inj}_i\ v$$
$$e ::= v \mid e\ e \mid (e, e) \mid e.i \mid \textsf{inj}_i\ e \mid$$
$$\textsf{case}\ e\ \textsf{of inj}_1\ x \Rightarrow e \mid \textsf{inj}_2\ x \Rightarrow e \mid$$
$$\textsf{newch} \mid \textsf{send}(e, e) \mid \textsf{tryrecv}\ e \mid \textsf{fork}\ e$$
$$K ::= [] \mid \textsf{send}(K, e) \mid \textsf{send}(v, K) \mid \textsf{tryrecv}\ K \mid \cdots$$

Derived forms

$$\textsf{None} \triangleq \textsf{inj}_1\ () \qquad \textsf{Some}(e) \triangleq \textsf{inj}_2\ e$$

Machine states

$$M \in \mathit{Bag} \quad \text{(finite bags of values)}$$
$$C \in \mathit{Chan} \overset{\text{fin}}{\rightharpoonup} \mathit{Bag}$$
$$T \in \mathbb{N} \overset{\text{fin}}{\rightharpoonup} \mathit{Exp}$$

Pure reduction (omitted) $\boxed{e \overset{\text{pure}}{\to} e'}$

Per-thread reduction $\boxed{C; e \to C'; e'}$

$$C; e \to C; e' \quad \text{when } e \overset{\text{pure}}{\to} e'$$
$$C; \textsf{newch} \to C[c \mapsto \emptyset]; c$$
$$C[c \mapsto M]; \textsf{send}(c, v) \to C[c \mapsto M \uplus \{v\}]; ()$$
$$C[c \mapsto \emptyset]; \textsf{tryrecv}\ c \to C[c \mapsto \emptyset]; \textsf{None}$$
$$C[c \mapsto M \uplus \{v\}]; \textsf{tryrecv}\ c \to C[c \mapsto M]; \textsf{Some}(v)$$

Machine reduction $\boxed{C; T \to C'; T'}$

$$\frac{C; e \to C'; e'}{C; T[i \mapsto K[e]] \to C'; T[i \mapsto K[e']]}$$
$$C; T[i \mapsto K[\textsf{fork}\ e]] \to C; T[i \mapsto K[()], j \mapsto e]$$

Figure 2. A language with asynchronous channels. (We denote the disjoint union $f \uplus [x \mapsto y]$ by $f[x \mapsto y]$.)

To demonstrate the effectiveness of abstraction and modular reasoning in Iris, we have applied it to a significant case study that builds a stack of abstractions, with each layer exporting only an abstract atomic specification to the layers above (Figure 1). The bottom two layers are physically atomic, while the top three are logically atomic.

At the bottom of the stack we start with a large-footprint physically atomic specification of channels in a λ-calculus with asynchronous message passing (§2). On top of that we first develop small-footprint specs (§5.2), followed by channels with a logically atomic blocking receive operation (§7.2). We then show that we can use channels (via a standard encoding [26]) to implement a logically atomic spec for mutable references, which we finally use in the verification of a fine-grained elimination stack ADT (§8). Put together, this constitutes a modular verification of elimination stacks running on a message-passing machine, performed completely within the Iris logic. Full details of the case study, as well as all other omitted technical details, appear in our technical appendix [1].

2. Iris – Part I: Monoids

Iris is a higher-order separation logic parameterized by the language of program expressions that one wishes to reason about.

For the purpose of this paper, we instantiate the programming language to the one shown in Figure 2, which provides primitive operations $\textsf{send}(e, e)$ and $\textsf{tryrecv}\ e$ for asynchronous (non-blocking) message passing via channels. We have chosen this language, rather than the usual heap-manipulating command language one finds in most separation-logic papers, in order to emphasize (1) that nothing about Iris is fundamentally tied to shared-memory concurrency, and (2) that heap-like abstractions can in fact be built up within the logic (§8).

The logic includes the usual connectives and rules of higher-order separation assertion logic as generated by the grammar below.

$$\Sigma ::= 1 \mid \textsf{Exp} \mid \textsf{Val} \mid \textsf{Prop} \mid \Sigma \times \Sigma \mid \Sigma \to \Sigma \mid \cdots$$
$$t, P, \varphi ::= () \mid (t, t) \mid \pi_1\ t \mid \pi_2\ t \mid \lambda x : \Sigma.\ t \mid \varphi\ t$$
$$\mid t =_\Sigma t \mid \textsf{False} \mid \textsf{True} \mid P \wedge P \mid P \vee P \mid P \Rightarrow P$$
$$\mid \exists x : \Sigma.\ P \mid \forall x : \Sigma.\ P \mid P * P \mid P \mathrel{-\!\!*} P \mid \cdots$$

Here, Σ denotes the type of a term (written t, P, or φ among others) in the logic. Terms include language expressions and values, propositions, and products and function spaces over these. The

$$\{\lfloor C \rfloor\}\ \textsf{newch}\ \{c.\ \lfloor C[c \mapsto \emptyset] \rfloor\}$$
$$\{\lfloor C[c \mapsto M] \rfloor\}\ \textsf{send}(c, m)\ \{v.\ v = () \wedge \lfloor C[c \mapsto M \uplus \{m\}] \rfloor\}$$
$$\{\lfloor C[c \mapsto M \uplus \{m\}] \rfloor\}\ \textsf{tryrecv}\ c\ \{v.\ v = \textsf{Some}(m) \wedge \lfloor C[c \mapsto M] \rfloor\}$$
$$\{\lfloor C[c \mapsto \emptyset] \rfloor\}\ \textsf{tryrecv}\ c\ \{v.\ v = \textsf{None} \wedge \lfloor C[c \mapsto \emptyset] \rfloor\}$$

Figure 3. Rules derived from the semantics of asynchronous channels (see Figure 2).

typing rules are standard and omitted (see the appendix [1]). The judgment $\Gamma \mid \mathcal{P} \vdash Q$ says that for all instantiations of the variables in Γ, the propositions $\mathcal{P}$ entail Q. The contexts are often left implicit. In the following sections, we will gradually introduce and motivate key features of the logic along with their associated terms and types.

Physical state. To specify and reason about the behavior of programs, the logic features Hoare triples, written $\{P\}\ e\ \{v.\ Q\}$. Here v serves as a binder for the return value in postcondition Q.

The logic is parametric in the syntax and semantics of the underlying programming language. As such, the logic only features basic structural rules and lacks Hoare axioms for primitive expressions. However, the logic provides a canonical way of adding such axioms, by internalizing the reduction semantics of the underlying language as axioms about the entire physical machine state.

To express this, Iris features a *physical state assertion*, written $\lfloor \varsigma \rfloor$, and a type $\textsf{State}$ of physical states of the underlying language. For every term ς of type $\textsf{State}$, $\lfloor \varsigma \rfloor$ asserts exclusive ownership of the physical state, along with the knowledge that the physical state is exactly ς. Using this assertion we can directly translate the reduction relation defined in Figure 2 into the axioms given in Figure 3. The physical state assertion only supports reasoning about the entire physical state; however, as we will see in §5.2, through the combination of monoids and invariants, it is possible to define more local reasoning principles.

Figure 4 summarizes the syntax and proof rules for physical states and selected structural Hoare rules. For now, ignore the $\Rrightarrow$ in CSQ and the $\Box$ in VSIMP. Combined, these rules simply yield the standard rule of consequence with implication. In the next section, we will explain how Iris generalizes that rule.

Ghost state. In addition to physical state assertions, Iris also supports assertions about *ghost state* (aka auxiliary state). Ghost state was originally proposed as a way to abstractly characterize some

Syntax

$$P ::= \cdots \mid P \Rrightarrow P \mid \{P\}\ e\ \{\varphi\} \mid \lfloor\varsigma\rfloor \mid \Box P \qquad \Sigma ::= \cdots \mid \mathsf{State}$$

Physical state axioms

$$\lfloor\varsigma\rfloor * \lfloor\varsigma'\rfloor \Rightarrow \mathsf{False}$$

Structural Hoare rules

$$\textsc{Frame}\quad \frac{\{P\}\ e\ \{v.\ Q\}}{\{P * R\}\ e\ \{v.\ Q * R\}} \qquad \textsc{Ret}\quad \{\mathsf{True}\}\ w\ \{v.\ v = w\}$$

$$\textsc{Bind}\quad \frac{\{P\}\ e\ \{v.\ Q\} \qquad \forall v.\ \{Q\}\ K[v]\ \{w.\ R\}}{\{P\}\ K[e]\ \{w.\ R\}} \qquad \textsc{VSImp}\quad \frac{\Box(P \Rightarrow Q)}{P \Rrightarrow Q}$$

$$\textsc{Csq}\quad \frac{P \Rrightarrow P' \qquad \{P'\}\ e\ \{v.\ Q'\} \qquad \forall v.\ Q' \Rrightarrow Q}{\{P\}\ e\ \{v.\ Q\}}$$

Figure 4. Physical state and selected structural rules.

knowledge about the history of a computation that is essential to verifying it [29]. More generally, ghost state is useful for modularly describing a thread's *knowledge* about some shared state, as well as the *rights* it has to modify it [7, 22].

In many logics, ghost state takes the form of a ghost heap. However, as the name implies, ghost state is a purely logical construct, introduced solely for the purpose of verification, and thus there is no need to tie it to a particular language or machine model. Consequently, in Iris, following several recent logics [22, 9, 24], we model ghost state as a *partial commutative monoid (PCM)*, which is taken as a parameter of the logic. We call the elements of this PCM *ghost resources*, and we use the term "resources" to refer both to physical states and ghost resources. (Assertions in Iris represent predicates over both kinds of resources.)

We represent the partial commutative monoid as a (total) commutative monoid with a zero element. The term $a \cdot b$ is thus always well formed. Formally, we require a set $|M|$ with two distinguished elements $\bot$ (zero, undefined) and ϵ (unit) and an operation $\cdot$ (compose) such that the following properties hold:

$$a \cdot b = b \cdot a \qquad \epsilon \cdot a = a \qquad (a \cdot b) \cdot c = a \cdot (b \cdot c)$$

$$\bot \cdot a = \bot \qquad \bot \neq \epsilon$$

We call $|M|$ the carrier of the monoid $M = (|M|, \bot, \epsilon, \cdot)$ and use $|M|^+$ to refer to the carrier without the zero element: $|M| \setminus \{\bot\}$.

To reason about these monoid elements, the logic features a type Monoid of monoid elements and internalizes the composition operation as a function on Monoid. For each term a of type Monoid, the logic features a corresponding ghost assertion $\boxed{a}$, which asserts $a \neq \bot$, along with ownership of an a fragment of the *global* ghost state. The global ghost state is the composition of all locally owned fragments. Ghost resources can thus be split and combined arbitrarily according to the chosen $\cdot$ operation: $\boxed{t \cdot u} \Leftrightarrow \boxed{t} * \boxed{u}$.

Since the ghost state is unrelated to the underlying physical state, we can update the current global ghost state arbitrarily at any time. We express updates of ghost state using *view shifts*. A view shift from P to Q, written $P \Rrightarrow Q$, asserts that it is possible to update the state from P to Q without changing the underlying physical state. The rule of consequence Csq in Figure 4 thus allows view shifts (and not just implications) to be applied in pre- and postconditions.

If we own a fragment a of the global ghost state, we must ensure that any updates of the ghost state are consistent with any fragments potentially owned by the environment (*e.g.,* other threads). The FpUpd rule in Figure 5 expresses that we can update a ghost fragment a to an element $b \in B$ if this update is frame-preserving ($a \rightsquigarrow B$);

Syntax

$$a, P ::= \cdots \mid \boxed{a} \mid a \cdot a \qquad \Sigma ::= \cdots \mid \mathsf{Monoid}$$

Ghost resource axioms

$$\boxed{a} * \boxed{b} \Leftrightarrow \boxed{a \cdot b} \qquad \mathsf{True} \Rightarrow \boxed{\epsilon} \qquad \boxed{\bot} \Rightarrow \mathsf{False}$$

Ghost resource updates

$$\textsc{FpUpd}\quad \frac{a \rightsquigarrow B}{\boxed{a} \Rrightarrow \exists b \in B.\ \boxed{b}} \qquad \begin{array}{c}\text{where } a \rightsquigarrow B \text{ is shorthand for}\\ \Box\forall a_f.\ a \# a_f \Rightarrow \exists b \in B.\ b \# a_f\end{array}$$

Figure 5. Syntax and proof rules for ghost resources.

that is, if it preserves an arbitrary composable frame a_f:

$$a \rightsquigarrow B \triangleq \Box\forall a_f.\ a \# a_f \Rightarrow \exists b \in B.\ b \# a_f$$
$$a \rightsquigarrow b \triangleq a \rightsquigarrow \{b\}$$

where $a \# a_f$ is notation for $a \cdot a_f \neq \bot$.

Note that the premise of rule FpUpd is merely a "fact": it does not express ownership of any state. To express such facts, the logic features an *always* modality, written $\Box P$, for asserting that P holds and does not assert any ownership. Such assertions are called *pure* and can be freely duplicated (*i.e.,* we have $\Box P \Rightarrow \Box P * \Box P$ and $\Box P \Rightarrow P$). View shifts and Hoare triples are other examples of pure assertions: the knowledge that a view shift or Hoare triple holds can thus be freely duplicated.

3. Monoid constructions

The monoid representing ghost resources—together with the frame-preserving update rules that it supports—can be seen as a way of *expressing a protocol* on logical state: given ownership of a particular monoid element, what does the owner know about the global ghost state and how are they permitted to update it? In this section, we present a number of useful monoid constructions (and their attendant frame-preserving update rules), representing different types of protocols. These protocols are not (yet) related to anything else; they just exist on their own. We will see in §5 how to *enforce* a protocol governing some other shared state.

The main novelty in this section is an encoding of *State Transition Systems* as a monoid. State Transistion Systems (STSs) provide a general and intuitive way of describing possible interference, expressed as a directed graph. The nodes of the graph represent possible ghost states, while edges describe how the ghost state may evolve. In §3.7, we show how to internalize this way of describing interference in Iris, through a general STS-as-monoid construction.

Recall that we represent PCMs as commutative monoids with zero. When defining monoid carriers, we leave this zero element $\bot$ implicit. Furthermore, we will not explicitly define the cases of the composition function that involve ϵ or $\bot$.

3.1 The exclusive monoid

The exclusive monoid over a set X supports two notions of ownership: exclusive ownership of an element $x \in X$ vs. no ownership. The owner of an element $x \in X$ thus has exact knowledge about the state. Formally, $\mathrm{Ex}(X)$ is the monoid with carrier $X \uplus \{\epsilon\}$ and composition only defined when one of the operands is ϵ.

This gives rise to the following frame-preserving update, which captures the ownership intuition given above.

$$x \rightsquigarrow a$$

Any non-trivial element of the monoid can be updated to any element. In terms of separation-logic reasoning, the already described physical assertions $\lfloor\varsigma\rfloor$ behave like ghost assertions about an exclusive monoid over physical states ς (see the axiom in Figure 4), except that of course one cannot use view shifts to update them.

3.2 The fractional monoid

It is often desirable to share knowledge about some piece of ghost state (*e.g.,* between different threads). A simple way to do that would be to reuse the carrier from $\textsc{Ex}(X)$, but let $a \cdot a = a$. This captures the right idea of knowledge, but gives up any notion of ownership: since all elements are duplicable, no frame-preserving update is possible. Instead, we would like a way to keep track of how much the knowledge about ghost state has spread, so that, after gathering it all up again, we can do a frame-preserving update.

This is achieved by the monoid $\textsc{Frac}(X)$ which has carrier $\big((0,1] \cap \mathbb{Q}\big) \times X \uplus \{\epsilon\}$ and composition

$$(q, x) \cdot (q', x') \triangleq (q + q', x) \qquad \text{if } q + q' \leq 1 \text{ and } x = x'$$

With this monoid, we can do any frame-preserving update *after collecting all the pieces*:

$$(1, x) \rightsquigarrow a$$

3.3 The product monoid

Using a product construction, we can combine any family of monoids $(M_i)_{i \in I}$ into a single monoid, while maintaining their individual reasoning principles (*i.e.,* frame-preserving updates). We define the carrier of the product monoid $\prod_{i \in I} M_i$ to be the product of the monoid carriers $\prod_{i \in I} |M_i|^+$. Composition is defined pointwise, if all of the constituent compositions are well defined (otherwise it is $\bot$).

We would like to operate on the ith component of this product monoid just as we would act on the individual monoid M_i, and indeed the following frame-preserving update rule holds.

$$a \rightsquigarrow_{M_i} B \;\vdash\; f[i \mapsto a] \rightsquigarrow \{f[i \mapsto b] \mid b \in B\}$$

3.4 Finite partial functions

A very common monoid in separation logics is the *heap*, modeled as a finite partial function from locations to values. We can obtain this monoid from the product monoid, by adding just one new piece.

Given a countably infinite domain X and a monoid (codomain) M, define $\textsc{FpFun}(X, M)$ to be the product monoid $\prod_{x \in X} M$, with the additional restriction of the carrier to elements f where the set $\mathrm{dom}(f) \triangleq \{x \mid f(x) \neq \epsilon_M\}$ is finite. This is well defined, since the set of these f contains the unit (which is the function mapping everything to ϵ_M) and is closed under composition. You can think of these f as finite partial functions to $|M|^+ \setminus \{\epsilon\}$, where the elements outside their domains are mapped to ϵ.

Since the domain of $f \in \textsc{FpFun}(X, M)$ is finite, an additional frame-preserving update becomes possible: a new element can be allocated.

$$\textsc{FpFunAlloc}$$
$$a \in |M|^+ \;\vdash\; f \rightsquigarrow \{f[x \mapsto a] \mid x \notin \mathrm{dom}(f)\}$$

Note that x is bound by the set comprehension: in applying this update, one cannot choose which x one gets.

Consider the monoid $\textsc{Heap} \triangleq \textsc{FpFun}(\mathit{Loc}, \textsc{Ex}(\mathit{Val}))$, where *Loc* is a set of locations. The carrier of this monoid consists of finite partial functions from locations to values, and composition of h and h' is defined if all pointwise compositions are defined. From the composition on $\textsc{Ex}(\mathit{Val})$, it follows that this is the case iff $\mathrm{dom}(h)$ and $\mathrm{dom}(h')$ are disjoint. This is exactly the standard composition of heaps in separation logic!

Now consider $\textsc{FHeap}(\mathit{Loc}) \triangleq \textsc{FpFun}(\mathit{Loc}, \textsc{Frac}(\mathit{Val}))$. This models a fractional heap, another commonly used monoid in separation logics. It can be used as ghost heap. We define the syntactic sugar $x \overset{q}{\hookrightarrow} w$ to mean $\boxed{[x \mapsto (q, v)]}$ if $q \in (0, 1]$, and False otherwise. Then we can show:

$$\forall v.\ \mathsf{True} \Rrightarrow \exists x.\ x \overset{1}{\hookrightarrow} v \tag{1}$$

$$\forall x, q_1, q_2, v, w.\ x \overset{q_1}{\hookrightarrow} v * x \overset{q_2}{\hookrightarrow} w \Leftrightarrow x \overset{q_1+q_2}{\hookrightarrow} v * v = w \tag{2}$$

$$\forall x, q, v.\ x \overset{q}{\hookrightarrow} v \Rightarrow x \overset{q}{\hookrightarrow} v \land q \in (0, 1] \tag{3}$$

$$\forall x, v, w.\ x \overset{1}{\hookrightarrow} v \Rrightarrow x \overset{1}{\hookrightarrow} w \tag{4}$$

We can use (1) to allocate a new ghost heap cell. Knowledge about the value and ownership of the location can be split and combined using (2). This is useful, in particular, when combining two pieces of a cell that were handed out earlier: one can learn that they point to the same value. Rule (3) says that fractional assertions have well-formed fractions. Having full ownership, the value in the ghost heap can be updated using (4).

3.5 Named monoid instances and multiple monoids

In the previous section, we had to fix the global monoid to provide a particular derived construction. We may, however, want to use two different constructions in the same proof without requiring them to share their monoid. It is also useful to be able to obtain a fresh *instance* of a monoid at any time. As it turns out, all the tools we need to mitigate this are already at hand. Given a family of monoids $(M_i)_{i \in I}$, we use a combination of the product and finite partial function monoids to define a global monoid M as follows:

$$M \triangleq \prod_{i \in I} \textsc{FpFun}(\mathbb{N}, M_i)$$

The product construction allows us to use a different monoid for independent parts of the proof, while the finite partial function construction provides named instances of these monoids. For the remainder of the article, we assume Iris has been instantiated with the monoid M given above, constructed from a family of monoids $(M_i)_{i \in I}$, taken as a parameter.

We write $\boxed{a : M_i}^{\gamma}$ (or just $\boxed{a}^{\gamma}$ if M_i is clear from the context) for $\boxed{[i \mapsto [\gamma \mapsto a]]}$ when $a \in |M_i|^+$, and for False when $a = \bot_{M_i}$. The ghost resource $\boxed{a : M_i}^{\gamma}$ thus asserts ownership of a part a of the instance named γ of monoid M_i in the current state. From the rules for ghost resources (Figure 5) and the frame-preserving updates in §3.3 and §3.4, we can derive the following rules for allocating, updating, and combining named monoid ghost resources.

$$\textsc{NewGhost}\quad \frac{a \in |M_i|^+}{\mathsf{True} \Rrightarrow \exists \gamma.\ \boxed{a : M_i}^{\gamma}} \qquad \textsc{GhostUpd}\quad \frac{a \rightsquigarrow_{M_i} B}{\boxed{a : M_i}^{\gamma} \Rrightarrow \exists b {\in} B.\ \boxed{b : M_i}^{\gamma}}$$

$$\textsc{GhostEq}$$
$$\boxed{a : M_i}^{\gamma} * \boxed{b : M_i}^{\gamma} \Leftrightarrow \boxed{a \cdot b : M_i}^{\gamma}$$

3.6 The authoritative monoid

A common pattern in concurrent reasoning is to put "someone" in charge of owning the global, *authoritative* state of some ghost resource, while "everyone else" owns fragments of that resource together with the knowledge that their fragments are all contained within the authoritative state.

We can capture this pattern with the monoid $\textsc{Auth}(M)$ where

$$|\textsc{Auth}(M)|^+ \triangleq \left\{ (x, a) \;\middle|\; \begin{array}{l} x \in |\textsc{Ex}(|M|^+)|^+ \land a \in |M|^+ \land \\ (x = \epsilon_{\textsc{Ex}(|M|^+)} \lor a \leq_M x) \end{array} \right\}$$

So the monoid consists of pairs, where the right component behaves just like M and the left component asserts exclusive ownership of a (non-zero) element of M. We impose the additional restriction that if an authoritative element is present, it must be an extension of the current fragment ($a \leq_M x$, which is shorthand for $\exists b.\ a \cdot_M b = x$). Composition is defined pointwise, and undefined if it is not in the carrier—*i.e.,* if the combined fragment exceeds the combined authoritative element, or if two authoritative elements are present. Note that $(\epsilon_{\text{Ex}(|M|+)}, \epsilon_M)$ is the unit and asserts no ownership or knowledge whatsoever, but (ϵ_M, ϵ_M) asserts that the authoritative element is ϵ_M.

We write $\bullet\, x$ to assert full ownership (x, ϵ_M) and $\circ\, a$ to assert fragmental ownership $(\epsilon_{\text{Ex}(|M|+)}, a)$. For consistency, we write $\bullet\, x, \circ\, a$ for (x, a).

The frame-preserving update for this monoid says that, if we own the authoritative element and some fragment, then we can exchange that fragment for anything that's compatible with "the rest":

$$\textsc{AuthUpd}\quad M \text{ cancellative} \wedge b \# a_f \vdash (\bullet\, a \cdot a_f, \circ\, a) \rightsquigarrow (\bullet\, b \cdot a_f, \circ\, b)$$

The rule requires that M be *cancellative*, which means

$$\forall a_f, a, b.\ a_f \cdot a = a_f \cdot b \neq \bot \Rightarrow a = b$$

The monoids $\text{Ex}(X)$ and $\text{Frac}(X)$ are cancellative for any X, and monoid products preserve cancellativity.

Building on an example from §3.4, consider $\text{Auth}(\text{Heap})$. This monoid allows us to assert authoritative ownership of some complete heap $\bullet\, h$ as well as fragmental ownership of, *e.g.,* individual memory locations as $\circ\, \ell \mapsto v$. Since Heap is cancellative, we obtain the following frame-preserving updates.

$$\textsc{AHeapUpd}\quad (\bullet\, h \cdot \ell \mapsto v, \circ\, \ell \mapsto v) \rightsquigarrow (\bullet\, h \cdot \ell \mapsto w, \circ\, \ell \mapsto w)$$

$$\textsc{AHeapAdd}\quad \ell \notin \text{dom}(h) \vdash (\bullet\, h) \rightsquigarrow (\bullet\, h \cdot \ell \mapsto v, \circ\, \ell \mapsto v)$$

Owning $(\bullet\, h, \circ\, \ell \mapsto v)$, we can deduce that $h = h' \cdot \ell \mapsto v$ for some h' by the definition of the carrier. Moreover, we can add a new location ℓ to $\text{Auth}(\text{Heap})$ *at any point we wish*, provided we can show that ℓ is not allocated in the current authoritative heap. This contrasts sharply with the frame-preserving update FPFunAlloc for Heap, where we can never be sure which location we get—we just know it will be fresh.

3.7 STSs with tokens

In their logic CaReSL [34], Turon *et al.* show how to usefully characterize the possible interference in a concurrent computation using *State Transition Systems (STSs)* with *tokens* [35]. As suggested in §1.1, these mechanisms serve to express *irreversibility* of state change and the *rights* to make state changes. Concretely, an STS comes equipped with a set of states and transitions between them, as well as a set of tokens and a mapping from states to tokens. The transition relation enforces irreversibility by restricting which states are accessible from which other states. The tokens assigned to a particular state are "owned" by the STS, and can be picked up by any thread when transitioning to other states. Tokens must be *conserved*: when taking transitions, the (disjoint) union of the tokens owned locally and the tokens owned by the STS's current state cannot change. The tokens owned locally by a given thread thus serve to limit the rights of other threads to make certain transitions.

As an example, consider the possible states of a remote procedure call (RPC) between a client and a server. Initially, a call has been sent, and it is the server's turn to send back a reply. Only after that has happened can the client receive the reply and terminate.

The corresponding STS is given in Figure 6. The second state, Rx, contains the token Srv. This means that the token is owned by the STS if we are at this state. To satisfy conservation of tokens, a transition from Tx to Rx can only be performed by giving up Srv. Owning Srv thus limits the possible interference from the environment: nobody but the owner of Srv can make this transition. Thus, saying we are *at least* in state Tx and we own token Srv amounts to saying we are *exactly* in state Tx.

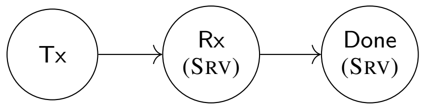

Figure 6. STS for a remote procedure call.

$$(s, T) \to (s', T') \triangleq s \to s' \wedge \mathcal{T}(s) \uplus T = \mathcal{T}(s') \uplus T'$$

$$\mathsf{frm}(s, T) \triangleq (s, \mathsf{TokSet} \setminus (\mathcal{T}(s) \uplus T))$$

$$\uparrow(S, T) \triangleq \{s' \in \mathcal{S} \mid \exists s \in S.\ \mathsf{frm}(s, T) \to^* \mathsf{frm}(s', T)\}$$

$$|\text{Sts}_{\mathcal{S}}| \triangleq \left\{ (s, S, T) \;\middle|\; \begin{array}{l} (s, S, T) \in \text{Ex}(\mathcal{S}) \times \mathcal{P}(\mathcal{S}) \times \mathcal{P}(\mathcal{T}) \wedge \\ (s = \epsilon \vee s \in S) \wedge \uparrow(S, T) = S \wedge \\ S \neq \emptyset \wedge \forall s \in S.\ \mathcal{T}(s) \cap T = \emptyset \end{array} \right\}$$

$$(s, S, T) \cdot (s', S', T') \triangleq (s \cdot s', S \cap S', T \uplus T')$$

(where composition is undefined if the result is not in $|\text{Sts}_{\mathcal{S}}|$)

Figure 7. Monoid encoding of STSs.

The monoid $\text{Sts}_{\mathcal{S}}$ for an STS $(\mathcal{S}, \to)$ with token set TokSet and token assignment $\mathcal{T} : \mathcal{S} \to \mathsf{TokSet}$ is defined in Figure 7. It has three parts. First is the STS's current authoritative state s. Ownership and knowledge of this state can only ever be held by one party—hence we use an exclusive monoid to represent it. Even if we do not own the authoritative s, we can use the second part of the monoid, S, to describe what we know about the set of *possible* states s could be, *e.g.,* that it is *accessible* from a certain state s_0. Composition on these sets is simply intersection, and knowledge about possible states is freely duplicable (as $S = S \cap S$). Finally, there is a set of locally owned tokens T: since we own them locally, we know that no other party owns them (composition is defined by $\uplus$), and that the protocol does not own them either ($\forall s \in S.\ \mathcal{T}(s) \cap T = \emptyset$).

Regarding the second part of the monoid, not all possible sets of states S are sensible. Every such set we define to exist will be *stable* knowledge, meaning it cannot be invalidated by actions of the environment. We thus restrict the set of possible states to be closed under transitions by the environment, as formalized by the upward-closure $\uparrow$ in Figure 7: $\uparrow(S, T) = S$. The figure also shows how to lift the relation $\to$ between states to a relation between state-token pairs: $(s, T) \to (s', T')$ says that, owning the tokens in T, one can move from s to s' and end up with the tokens in T'. In our RPC STS, for example, we have $(\mathsf{Tx}, \{\text{Srv}\}) \to (\mathsf{Rx}, \emptyset)$. On the other hand, $(\mathsf{Tx}, \emptyset)$ has no successor.

This definition gives rise to the following frame-preserving update. Owning the authoritative state and some tokens, we can make any transition that is justified by the tokens we own:

$$\textsc{StsUpd}\quad \frac{(s, T) \to^* (s', T')}{(s, S, T) \rightsquigarrow (s', \uparrow(\{s'\}, T'), T')}$$

Note how we use $\uparrow$ to stabilize knowledge—having transitioned to s', we know that we are *at least* in s'.

While encoding STSs as monoids is interesting, we can only unleash its true power—recovering CaReSL's workhorse, the "island update" rule—once we have a way of enforcing STSs as protocols over the evolution of some shared state. For that, we need invariants.

4. Iris – Part II: Invariants

So far, Iris only supports reasoning about state that is owned by one thread or another. There is no mechanism yet for sharing state between threads. To support such sharing, we extend the logic now with *invariants*. Given a proposition P, the assertion $\boxed{P}^{\iota}$ asserts the existence of a shared invariant named ι, which governs resources satisfying P. Since invariant assertions just assert the existence of a shared invariant, they are pure and thus freely duplicable: $\boxed{P}^{\iota} \Leftrightarrow \square\boxed{P}^{\iota}$.

To access the shared resource they govern, invariants can be *opened* for the duration of a physically atomic expression. Opening an invariant grants temporary ownership of the shared resource. Once the invariant is reestablished (possibly after resources have been transferred in or out), it can be *closed* again. The following Inv rule (which will turn out to be derivable from other rules) grants e ownership of shared resource R for the duration of its execution.

$$\textsc{Inv}\quad \frac{\{\triangleright R * P\}\ e\ \{v.\ \triangleright R * Q\}_{\mathcal{E}} \qquad e \text{ phys. atomic}}{\boxed{R}^{\iota} \vdash \{P\}\ e\ \{v.\ Q\}_{\mathcal{E} \uplus \{\iota\}}}$$

For now, ignore the *later* modality $\triangleright$, we will come back to it later. The rule (read backwards) says that, if you know that an invariant R with name ι exists, you can add its resources to the precondition. The verification of e can then freely use those resources as it wishes, but it must give back control of some resources satisfying R when it is done.

Masks. To ensure that each invariant is opened at most once, we annotate Hoare triples with an invariant mask. For the Hoare triple $\{P\}\ e\ \{v.\ Q\}_{\mathcal{E}}$ we can assume the invariants in $\mathcal{E}$ hold prior to the execution and must reestablish them after each single physical step. We say that the invariants in $\mathcal{E}$ are *enabled*, which means that they hold on some portion of the shared (physical and logical) state. Here $\mathcal{E}$ is a term of type $\mathsf{InvMask}$—the type of invariant masks, which are simply sets of invariant names.

Mask-changing view shifts. It is helpful to think about the Inv rule as combining three separate steps of reasoning. First, note that, in the conclusion, ι is enabled, which means that it can be opened (disabled), and the resource satisfying $\triangleright R$ can be transferred from shared control to the local control of e. Second, this resource, together with P, is used to reason about e, which reestablishes $\triangleright R$ and some additional Q under the assumption that ι is disabled. Third, the invariant is closed (reenabled), which returns the resource satisfying $\triangleright R$ to shared control and restores the original invariant mask. For reasons that will become clear in §7, we find it useful to be able to reason about the opening and closing steps of this rule independently of the reasoning about e that goes on in between. Consequently, instead of taking Inv as primitive, Iris employs a *novel notion of view shifts that can open or close invariants*.

This is achieved by annotating view shifts with two sets of invariants: those that are enabled *before* and *after* the update. The view shift $P\ {}^{\mathcal{E}_1}\!\!\Rrightarrow^{\mathcal{E}_2} Q$ expresses that it is possible to update the local state from P to Q without updating the physical state, where *only* the invariants in $\mathcal{E}_1$ and $\mathcal{E}_2$ are required to hold before and after the update, respectively. We write $P \Rrightarrow_{\mathcal{E}} Q$ as syntactic sugar for $P\ {}^{\mathcal{E}}\!\!\Rrightarrow^{\mathcal{E}} Q$. The InvOpen and InvClose rules in Figure 8 express invariant opening and closing as mask-changing view shifts.

The rule of atomic consequence (ACsq in Figure 9) allows us to open invariants $\mathcal{E}'$ for the duration of an expression e, provided we close them again afterwards. This rule can be composed with InvOpen and InvClose to obtain Inv as a derived rule.

Note that ACsq can only be sound for *physically atomic* expressions: After doing one step in e, another thread may be next to compute, and it may rely on invariants in $\mathcal{E}'$. We will see in §7 how to give specifications for functions that are *not physically atomic*, while still allowing opening invariants. Our more general notion of mask-changing view shifts will prove to be very helpful there.

View shift rules. View shifts permit a frame rule VSFrame similar to the usual frame rule for Hoare triples. However, this rule serves

Syntax

$$P, \iota, \mathcal{E} ::= \cdots \mid \boxed{P}^{\iota} \mid \triangleright P \qquad \Sigma ::= \cdots \mid \mathsf{Name} \mid \mathsf{InvMask}$$

Proof rules

$$\textsc{NewInv}\quad \mathsf{infinite}(\mathcal{E}) \vdash \triangleright P \Rrightarrow_{\mathcal{E}} \exists \iota \in \mathcal{E}.\ \boxed{P}^{\iota}$$

$$\textsc{InvOpen}\quad \boxed{P}^{\iota} \vdash \mathsf{True}\ {}^{\{\iota\}}\!\!\Rrightarrow^{\emptyset} \triangleright P \qquad\qquad \textsc{InvClose}\quad \boxed{P}^{\iota} \vdash \triangleright P\ {}^{\emptyset}\!\!\Rrightarrow^{\{\iota\}} \mathsf{True}$$

Figure 8. Syntax and proof rules for invariants.

$$\textsc{VSTrans}\quad \frac{\mathcal{E}_2 \subseteq \mathcal{E}_1 \cup \mathcal{E}_3 \qquad P\ {}^{\mathcal{E}_1}\!\!\Rrightarrow^{\mathcal{E}_2} Q \qquad Q\ {}^{\mathcal{E}_2}\!\!\Rrightarrow^{\mathcal{E}_3} R}{P\ {}^{\mathcal{E}_1}\!\!\Rrightarrow^{\mathcal{E}_3} R} \qquad \textsc{VSFrame}\quad \frac{P\ {}^{\mathcal{E}_1}\!\!\Rrightarrow^{\mathcal{E}_2} Q}{P * R\ {}^{\mathcal{E}_1 \uplus \mathcal{E}'}\!\!\Rrightarrow^{\mathcal{E}_2 \uplus \mathcal{E}'} Q * R}$$

$$\textsc{VSTimeless}\quad \frac{\mathsf{timeless}(P)}{\triangleright P \Rrightarrow P} \qquad \textsc{ACsq}\quad \frac{P\ {}^{\mathcal{E} \uplus \mathcal{E}'}\!\!\Rrightarrow^{\mathcal{E}} P' \qquad \{P'\}\ e\ \{v.\ Q'\}_{\mathcal{E}} \qquad \forall v.\ Q'\ {}^{\mathcal{E}}\!\!\Rrightarrow^{\mathcal{E} \uplus \mathcal{E}'} Q \qquad e \text{ phys. atomic}}{\{P\}\ e\ \{v.\ Q\}_{\mathcal{E} \uplus \mathcal{E}'}}$$

Figure 9. Most important view shift rules (cf. our appendix [1]).

not only to frame resources around a view shift. It can also be used to frame invariants: if some invariant ι not covered by either mask of the view shift is known to be enabled, it remains so when the view shift is applied. Hence, view shifts may only affect invariants that they explicitly name in one of their masks. The same kind of framing is possible for Hoare triples; we refer the reader to the appendix [1] for the full set of rules.

The view shift transitivity rule VSTrans allows two view shifts to be combined, provided they agree on the invariants that are enabled between the two view shifts and that those invariants are not forgotten in the conclusion ($\mathcal{E}_2 \subseteq \mathcal{E}_1 \cup \mathcal{E}_3$). This side condition is necessary to ensure soundness of the VSFrame rule.

New invariants are created by transferring local resources satisfying the invariant to the shared state. As this involves a relabeling of resources (from "local" to "shared") but not any actual change to physical state, it can be expressed as a view shift. The NewInv rule in Figure 8 allocates an invariant with a name ι, chosen from a set of possible names $\mathcal{E}$. We require $\mathcal{E}$ to be infinite to make sure that there is some invariant name in there that is not taken already. And since we can pick the infinite set, we can reason that the different invariants we create have different names (by creating them with applications of NewInv with disjoint $\mathcal{E}$'s). Disjointness of invariant names is important if we wish to apply Inv in nested fashion, since we will have to prove that each invariant is opened at most once.

Later modality. Since any Iris assertion can serve as an invariant, one can define invariants that refer to other invariants or even themselves. This *impredicativity* introduces a potential circularity. Following iCAP [33] and CaReSL [34], we use step-indexing to break this circularity, and internalize the notion of steps using a *later* modality. The assertion $\triangleright P$ expresses that P holds one step later. To ensure that invariants are well defined, the shared resource backing up $\boxed{P}^{\iota}$ need only satisfy $\triangleright P$. Thus, opening an invariant (InvOpen) grants ownership of the shared resource one step later. Conversely, to close an invariant (InvClose), it suffices to reestablish the shared resource one step later. If a resource P holds now, it also holds later: $P \Rightarrow \triangleright P$.

With *timeless* propositions, things are simpler. Timeless propositions are not affected by $\triangleright$, as expressed in rule VSTimeless. Examples of such propositions include physical state assertions and ghost assertions. Timelessness matters primarily when reasoning about propositions appearing in invariants: if an invariant is timeless, we can immediately view shift from $\boxed{P}^{\iota}$ to P by opening the invariant to obtain $\triangleright P$, then applying VSTIMELESS. The formal definition of timelessness can be found in the appendix [1].

5. Invariant constructions

In combination with invariants, the monoid constructions presented in §3 let us encode powerful patterns for local reasoning about protocols on shared resources, including several that were developed in previous work [34, 33, 8]. To achieve this, an abstract *protocol* (as defined by a monoid) is tied to an *interpretation* of what the protocol is intended to guarantee about some shared resources, and the desired connection between the two is enforced with an invariant. It is then possible to derive Hoare triples for expressions that update the shared resources in accordance with the protocol.

5.1 STSs with interpretation

The whole point of the encoding of STSs from §3.7 was to be able to define protocols capable of governing some shared resource (*e.g.,* the pointer structure implementing some concurrent mutable ADT, although in general the shared resource need not be physical). To support this functionality, we begin by extending the STSs from §3.7 with an interpretation $\varphi(s)$ for each state s, which will say what property should hold of the underlying shared resource when the current (*i.e.,* authoritative) state of the STS is s.

Let the STS $(\mathcal{S}, \rightarrow)$, interpretation function $\varphi : \mathcal{S} \rightarrow \mathsf{Prop}$, and instance name γ be given. We then define the following invariant:

$$\mathsf{STSInv}(\mathcal{S}, \varphi, \gamma) \triangleq \exists s.\ \boxed{(s, \mathcal{S}, \emptyset) : \textsc{Sts}_{\mathcal{S}}}^{\gamma} * \varphi(s)$$

By creating this invariant, we enforce that the current authoritative state s of the STS instance γ is a shared resource, as is whatever shared resource backs up its interpretation $\varphi(s)$, which is the state that multiple threads want to access concurrently.

Returning to the RPC example from §3.7, let us assume knowledge of an STS invariant governing an instance γ of the RPC STS in Figure 6. Suppose we own resources P and the STS resource $\boxed{(\mathsf{Rx}, \emptyset)}^{\gamma}$ and want to transition the STS to state Done and establish Q. Our ghost resource represents knowledge that the STS is at least in state Rx (and ownership of no tokens). When we open the invariant, we learn the authoritative state, $s \in \{\mathsf{Rx}, \mathsf{Done}\}$, and take ownership of $\triangleright\varphi(s)$. (The knowledge we started with ensures $s \neq \mathsf{Tx}$.) Since we want to transition to Done, our primary proof obligation is to show that, no matter what s is, we can update the resources on hand, $P * \triangleright\varphi(s)$, to $\triangleright\varphi(\mathsf{Done}) * Q$. Having proven that, we can then close the invariant and walk away with $\boxed{(\mathsf{Done}, \emptyset)}^{\gamma} * Q$. (The ghost represents our updated knowledge about the STS.)

This general reasoning pattern is reflected in the derived rule STS in Figure 10. If we ignore the tokens (for simplicity), the rule says: if we know the STS instance γ is at least in state s_0 and want to update the STS, then it suffices to show that for any future state s of s_0 we can, adding $\triangleright\varphi(s)$ to our precondition, establish $\triangleright\varphi(s')$ in our postcondition (for some future state s' of s).

The rule matches exactly CaReSL's island update rule [34]. As with INV from the previous section, it is derived using ACSQ, and the proof follows the same decomposition: open invariant, reason with additional resources, close invariant. The only difference is an additional frame-preserving update using STSUPD right before closing the invariant.

5.2 Authoritative monoids with interpretation

In this section, we implement the second layer of our stack (Figure 1) using reasoning principles similar to those Krishnaswami *et al.* developed for "superficially substructural types" [22]. We derive these principles by marrying an interpretation to the monoid $\textsc{Auth}(M)$, much as we did for the monoid $\textsc{Sts}_{\mathcal{S}}$ in §5.1.

Consider the Hoare triples we obtained in Figure 3 to reason about language primitives. They all carry in their precondition a physical assertion, $\lfloor\varsigma\rfloor$, about the global machine state, which can only ever be held by a single party. These are called *large-footprint specifications*, as opposed to the *small-footprint specifications* given in Figure 11, which only mention the channel they operate on [31]. We aim to derive these small-footprint specs. We will achieve this by putting ownership of the entire physical state into an invariant, so that it is shared by everybody. The invariant ties this physical state to a ghost resource so that fragments of the ghost resource (which can be split up among different threads) effectively control fragments of the physical state.

We can model the state of a network using the monoid $\textsc{Net} \triangleq \textsc{FpFun}(\mathsf{Chan}, \textsc{Ex}(\mathsf{Bag}))$. This gives us an adequate level of sharing: we can make assertions about individual channels, without mentioning all the others. Next, we define the interpretation $\varphi : |\textsc{Net}|^{+} \rightarrow \mathsf{Prop}$ that we wish to hold for the *authoritative* state of this monoid, *i.e.,* the composition of all fragments:

$$\varphi(C) \triangleq \lfloor C \rfloor$$

(Here, we implicitly coerce between the finite partial functions in $|\textsc{Net}|^{+}$, and the ones comprising the possible physical states).

To tie the two together, we use an instance γ of $\textsc{Auth}(\textsc{Net})$ and the following invariant.

$$\mathsf{ChanInv} \triangleq \exists C.\ \boxed{\bullet\, C : \textsc{Auth}(\textsc{Net})}^{\gamma} * \varphi_{\perp}(C)$$

We extend φ to $\varphi_{\perp} : |\textsc{Net}| \rightarrow \mathsf{Prop}$ by setting $\varphi_{\perp}(\perp) = \mathsf{False}$.

Now assume that we own $\boxed{\circ\, c \mapsto M}^{\gamma}$ (which we will write $c \prec M$), and consider what we obtain upon opening the invariant ChanInv. While we cannot know which C witnesses the existential, we know from the definition of $\textsc{Auth}(\textsc{Net})$ that $c \mapsto M \leq C$, and thus $C(c) = M$. With our temporary ownership of the shared physical state $\lfloor C \rfloor$, we can justify operating on our channel c. All of this follows just from owning $c \prec M$, which is a purely logical assertion that has nothing to do *per se* with the physical state—it is the invariant which lets *us* decide how we want to tie the two together. After the operation on c, we make us of the fact that we own both the authoritative state and the fragment governing c, so we know nobody else can hold any knowledge about this channel. Hence we can do a frame-preserving update synchronizing the ghost state with the new physical state, and closing the invariant.

In general, we may want to give an interpretation $\varphi : |M|^{+} \rightarrow \mathsf{Prop}$ to every (non-zero) element of some cancellative monoid M. We can do so with the invariant

$$\mathsf{AuthInv}(M, \varphi, \gamma) \triangleq \exists c.\ \boxed{\bullet\, c : \textsc{Auth}(M)}^{\gamma} * \varphi_{\perp}(c)$$

The reasoning pattern enabled by this construction is codified by rule AUTH in Figure 10.

The rule says that if we know an invariant tying the authoritative part of γ to φ, and if we own the fragment a in γ, then we can gain access to the interpretation $\varphi_{\perp}(a \cdot a_f)$ of the current authoritative state (where a_f is everything owned by the environment). If, using the resources of the interpretation together with the P that we carried in, we can establish $\varphi_{\perp}(b \cdot a_f)$ (which implies $b \,\#\, a_f$) together with Q, then we are allowed to carry out whatever was left in Q, and update our ghost fragment to b. Like STS, this rule combines opening and closing an invariant with a frame-preserving update (except here the update uses AUTHUPD rather than STSUPD).

The triples given in Figure 11 are derived using AUTH. The mask $\mathcal{E}_{\mathsf{chan}}$ will contain the name of the invariant governing ChanInv. Note that we make crucial use of the VSTIMELESS rule here: After opening the invariant, we actually obtain $\triangleright\lfloor C \rfloor$. But physical state assertions are timeless, so we can immediately view shift that to

$$\textsc{Sts}\quad \frac{\forall s \in \uparrow(\{s_0\}, T).\ \{\triangleright\varphi(s) * P\}\ e\ \{v.\ \exists s', T'.\ (s,T) \rightarrow^* (s',T') * \triangleright\varphi(s') * Q\}_{\mathcal{E}} \qquad e \text{ phys. atomic}}{\boxed{\mathsf{STSInv}(\mathcal{S}, \varphi, \gamma)}^{\iota} \vdash \left\{\boxed{(s_0, T)}^{\gamma} * P\right\}\ e\ \left\{v.\ \exists s', T'.\ \boxed{(s', T')}^{\gamma} * Q\right\}_{\mathcal{E} \uplus \{\iota\}}}$$

$$\textsc{Inv}\quad \frac{\{\triangleright R * P\}\ e\ \{v.\ \triangleright R * Q\}_{\mathcal{E}} \qquad e \text{ phys. atomic}}{\boxed{R}^{\iota} \vdash \{P\}\ e\ \{v.\ Q\}_{\mathcal{E} \uplus \{\iota\}}}$$

$$\textsc{Auth}\quad \frac{\begin{array}{c} e \text{ phys. atomic} \qquad M \text{ cancellative} \\ \forall a_f.\ \{\triangleright\varphi_\perp(a \cdot a_f) * P\}\ e\ \{v.\ \exists b.\ \triangleright\varphi_\perp(b \cdot a_f) * Q\}_{\mathcal{E}} \end{array}}{\boxed{\mathsf{AuthInv}(M, \varphi, \gamma)}^{\iota} \vdash \left\{\boxed{a}^{\gamma} * P\right\}\ e\ \left\{v.\ \exists b.\ \boxed{b}^{\gamma} * Q\right\}_{\mathcal{E} \uplus \{\iota\}}}$$

Figure 10. Derived rules to reason about shared state.

$$\{\mathsf{True}\}\ \textbf{newch}\ \{c.\ c \prec \emptyset\}$$

$$\{c \prec M\}\ \textbf{send}(c, m)\ \{v.\ v = () \land c \prec M \uplus \{m\}\}_{\mathcal{E}_{\mathsf{chan}}}$$

$$\{c \prec M\}\ \textbf{tryrecv}\ c$$
$$\left\{\begin{array}{l} v.\ (M = \emptyset \land v = \mathsf{None} \land c \prec \emptyset) \lor \\ \quad (\exists m.\ m \in M \land v = \mathsf{Some}(m) \land c \prec M \setminus \{m\}) \end{array}\right\}_{\mathcal{E}_{\mathsf{chan}}}$$

Figure 11. Derived rules for language primitives (see Figure 3 for the basic rules).

$\lfloor C \rfloor$ and not bother with the $\triangleright$. Furthermore, we obtain a single unified description of **tryrecv** by case distinction on M.

This pattern turns out to be very expressive. It is applicable whenever it is necessary to collect some state in a central place, and useful to spread ownership and knowledge about parts of this state. For example, in a heap-manipulating language, this pattern easily scales to provide fractional permissions. Also note that Auth corresponds closely to Krishnaswami *et al.*'s "sharing rule" [22]. This is yet another example of Iris's ability to derive powerful reasoning principles that are built fixed into prior logics.

6. Semantics

The semantics of Iris is defined in the accompanying technical appendix and formalized in the accompanying Coq development [1]. In this section we give a very brief overview of the model.

To model invariants, assertions in Iris are modeled relative to a *world* that describes the invariants allocated so far. Since invariants are themselves expressed as assertions, this introduces a circularity in the modeling of assertions and worlds. This is the standard type-world circularity that also arises in models of type systems with dynamic allocation and higher-order store [3]. We use standard metric-based techniques to solve the circularity [4, 6] and, in particular, Sieczkowski *et al.*'s library [32] for solving such circularities in Coq. Crucially, the construction of the semantic domain of assertions—which allows us to model invariants—is parametric in the ghost state monoid. Invariants and monoids are thus also orthogonal semantically.

Iris's adequacy theorem expresses that if $\{\lfloor\varsigma\rfloor\}\ e\ \{v.\ \varphi(v)\}$ is derivable and e executes to a value v when started in physical state ς, then v satisfies φ. Here, $\varphi(v)$ is a pure predicate, *i.e.,* it describes a property of v and cannot mention resources or invariants. Formally:

Theorem 1 (Adequacy)**.**
If $\vdash \{\lfloor\varsigma\rfloor\}\ e\ \{v.\ \varphi(v)\}$ *and* $\varsigma; [i \mapsto e] \rightarrow^* \varsigma'; [i \mapsto v] \uplus T'$,
then $\varphi(v)$.

The T' in the final state permits e to fork off other threads.

Soundness of the underlying higher-order separation logic, the rules in Figures 4, 5, 8, and 9, and the adequacy theorem have all been proven in Coq [1].

$$\{\mathsf{True}\}\ \textbf{newch}\ \{c.\ c \prec \emptyset\}$$

$$\langle M.\ c \prec M\rangle\ \textbf{send}(c, m)\ \langle v.\ c \prec M \uplus \{m\} \land v = ()\rangle^{\mathcal{E}_{\mathsf{chan}}}$$

$$\langle M.\ c \prec M\rangle\ \mathsf{recv}\ c\ \langle m.\ c \prec M \setminus \{m\} \land m \in M\rangle^{\mathcal{E}_{\mathsf{chan}}}$$

where

```
recv ≜ rec recv(c).
           let v = tryrecv c in
           case v of None ⇒ recv c | Some(m) ⇒ m
```

Figure 12. Logically atomic spec for channels with blocking recv.

7. Logical atomicity

How can triples like the ones in Figure 11 be used? Of course, one could just use them as normal Hoare triples, and establish $c \prec M$ before calling them. This, however, would effectively sequentialize access to the channel: every caller would have to prove that they exclusively own the channel in order to access it.

Moreover, we often *want* several threads to be able to "race" for access to the resource. For example, consider the case where there is an invariant ι_{even} governing the channel, making sure it only ever contains even numbers:

$$\boxed{\exists M.\ c \prec M \land \forall m \in M.\ m \in \mathbb{N} \land m \text{ is even}}^{\iota_{\mathsf{even}}}$$

Since **send** is a *physically atomic* operation, we can use Inv to gain access to the channel. We can open the invariant *around* the call to **send**, atomically observing the current state of the channel. If we are sending an even number, we can reestablish the invariant after **send** is done, and close it again. This is sound because no other thread can interfere with the physically atomic call to **send**. In this case, we say that we have a *physically atomic triple* for **send**.

Now consider the blocking implementation of recv defined in Figure 12. We would like to do the same kind of reasoning, *e.g.,* to verify that recv c always returns an even number. But recv is not physically atomic, so Inv does not apply. However, intuitively, recv *behaves as if it were atomic*: there is a single point in time (often called the *linearization point*) where the receiving action is (logically) committed, namely the instant when **tryrecv** succeeds. We ought to be able to exploit this, and call recv based on the channel assertion that is governed by ι_{even}—but if we only have a normal Hoare triple for recv, there is no way to do this. Our goal is thus to find a notion of a *logically atomic triple* that admits the reasoning principles given in Figure 10, but is applicable to functions like recv.

Now, note that all of our reasoning principles in Figure 10 were derived using ACsq, which lets us compose physically atomic triples with view shifts *that open and close invariants*, whereas the corresponding rule for general expressions, Csq, works only for view shifts with the same set of invariants enabled on both sides. It is thus justified to say that ACsq *logically captures the essence of what it means to be atomic*. Hence, logically atomic triples should support the same kind of reasoning, and we will use this insight in determining how to define them.

7.1 Logically atomic triples

In order to motivate the definition of logical atomic specifications, it is helpful to consider the needs of both a client *using* such a specification, and a module *proving* it.

Client perspective. What would it mean for recv to support a reasoning principle like ACSQ? Well, for example, we would like to have the following instantiation of ACSQ:

$$\frac{\begin{array}{c}\{c \prec M\}\ \mathsf{recv}\ c\ \{m.\ c \prec M \setminus \{m\} \land m \in M\}_{\mathcal{E}_{\mathsf{chan}}} \\ P\ {}^{\mathcal{E}_{\mathsf{chan}} \uplus \mathcal{E}'}\!\!\Rrightarrow^{\mathcal{E}_{\mathsf{chan}}}\ c \prec M \\ \forall m.\ c \prec M \setminus \{m\} \land m \in M\ {}^{\mathcal{E}_{\mathsf{chan}}}\!\!\Rrightarrow^{\mathcal{E}_{\mathsf{chan}} \uplus \mathcal{E}'}\ Q(m)\end{array}}{\{P\}\ \mathsf{recv}\ c\ \{m.\ Q(m)\}_{\mathcal{E}_{\mathsf{chan}} \uplus \mathcal{E}'}} \tag{5}$$

As a client of the module implementing recv , we would expect that module to prove the first premise. We would then give proofs for the two view shifts, and obtain the conclusion.

Now, recv is not physically atomic, so (5) is not sound. However, we can transform (5) into the "elimination form" of its first premise, *i.e.,* into the statement that ACSQ can be applied to recv:

$$\begin{aligned}&\forall P, Q, \mathcal{E}', M.\ (P\ {}^{\mathcal{E}_{\mathsf{chan}} \uplus \mathcal{E}'}\!\!\Rrightarrow^{\mathcal{E}_{\mathsf{chan}}}\ c \prec M) \land \\ &\quad (\forall m.\ c \prec M \setminus \{m\} \land m \in M\ {}^{\mathcal{E}_{\mathsf{chan}}}\!\!\Rrightarrow^{\mathcal{E}_{\mathsf{chan}} \uplus \mathcal{E}'}\ Q(m)) \\ &\quad \Rightarrow \{P\}\ \mathsf{recv}\ c\ \{v.\ Q(v)\}_{\mathcal{E}_{\mathsf{chan}} \uplus \mathcal{E}'}\end{aligned} \tag{6}$$

If *this* is the interface provided by the module, then we as client can do exactly the reasoning that (5) would allow us to do! The key observation is that such a statement can be made about any operation, be it physically atomic or not.

There are still some problems, though. Suppose we want to call recv on the channel c governed by the invariant ι_{even}. To do this, we have to choose some M and then prove that we can view shift from P to $c \prec M$. However, we only learn the current M *after* opening the invariant (*inside* the proof of the the view shift), so we have no way to fix it *a priori*! Furthermore, we cannot prove the second view shift: at that point, we have forgotten that all messages in M are even numbers, so we cannot reestablish the invariant.

To solve this, we introduce a special treatment for M such that it only has to be fixed after opening the invariant. We will also bake in an application of FRAME, similar to how ACSQ was baked into (6). This leads us to the following:

$$\begin{aligned}&\forall P, Q, R, \mathcal{E}'.\ (P\ {}^{\mathcal{E}_{\mathsf{chan}} \uplus \mathcal{E}'}\!\!\Rrightarrow^{\mathcal{E}_{\mathsf{chan}}}\ \exists M.\ c \prec M * R(M)) \land \\ &\quad \begin{pmatrix}\forall M, m.\ c \prec M \setminus \{m\} \land m \in M * R(M) \\ {}^{\mathcal{E}_{\mathsf{chan}}}\!\!\Rrightarrow^{\mathcal{E}_{\mathsf{chan}} \uplus \mathcal{E}'}\ Q(M, m)\end{pmatrix} \\ &\quad \Rightarrow \{P\}\ \mathsf{recv}\ c\ \{m.\ \exists M.\ Q(M, m)\}_{\mathcal{E}_{\mathsf{chan}} \uplus \mathcal{E}'}\end{aligned} \tag{7}$$

This is strong enough to be useful in our example. We choose:

$$P \triangleq \mathsf{True} \qquad Q(\_, m) \triangleq m \in \mathbb{N} \land m \text{ is even} \qquad \mathcal{E}' \triangleq \{\iota_{\mathsf{even}}\}$$

$$R(M) \triangleq \forall m \in M.\ m \in \mathbb{N} \land m \text{ is even}$$

The view shifts are then easy to show. Applying (7), this yields the desired triple, stating that recv c always returns an even number:

$$\{\mathsf{True}\}\ \mathsf{recv}\ c\ \{m.\ m \in \mathbb{N} \land m \text{ is even}\}_{\mathcal{E}_{\mathsf{chan}} \uplus \{\iota_{\mathsf{even}}\}}$$

Note how our ability to prove the postcondition (that m is even) relies crucially on the fact that the frame R can depend on M.

Module perspective. Let us now switch roles, and consider the case of a module that wants to *provide* a specification like (7). This means we get to *assume* some arbitrary P, Q, R, $\mathcal{E}'$ and the associated view shifts, and we have to prove an (ordinary) Hoare triple. Unlike in the normal case of proving a triple, we do not have access to the resource $c \prec M$ that we operate on. Instead, we own some P, about which we only know that we can view shift to the desired resource. How can that be enough?

$$\begin{aligned}&\langle x.\ \alpha\rangle\ e\ \langle v.\ \beta\rangle^{\mathcal{E}_M}_{\mathcal{E}} \triangleq\ \mathcal{E}_M \# \mathcal{E} \land \\ &\quad \Box\begin{pmatrix}\forall P, Q, R, \mathcal{E}_R. \\ \quad \langle x.\ P \Leftrightarrow \alpha \mid R(x), \mathcal{E}_R \mid v.\ \beta \Rrightarrow Q(x, v)\rangle^{\mathcal{E}_M}_{\mathcal{E}} \\ \quad \Rightarrow \{P\}\ e\ \{v.\ \exists x.\ Q(x, v)\}_{\top}\end{pmatrix} \\ &\langle x.\ P \Leftrightarrow \alpha \mid R, \mathcal{E}_R \mid v.\ \beta \Rrightarrow Q\rangle^{\mathcal{E}_M}_{\mathcal{E}} \triangleq \\ &\quad \mathsf{timeless}(P) \land \mathcal{E}_R \# \mathcal{E} \cup \mathcal{E}_M \land \\ &\quad (P\ {}^{-\mathcal{E}_M}\!\!\Leftrightarrow^{-\mathcal{E}_M - \mathcal{E}_R}\ \exists x.\ \alpha * R) \land \\ &\quad (\forall x, v.\ \beta * R\ {}^{-\mathcal{E}_M - \mathcal{E}_R}\!\!\Rrightarrow^{-\mathcal{E}_M}\ Q)\end{aligned}$$

Figure 13. Definition of logically atomic triples and atomic shifts.

The key is this: at the "linearization point" (as explained above, the single step of execution where the operation "commits"), we will make use of the view shifts we have been given, together with rule ACSQ, to grant us temporary access to $c \prec M$ for that one step. Formally, at the linearization point, we will use the view shift from P to $c \prec M$, which may open some islands $\mathcal{E}'$—hence, we call it the *opening view shift*. After operating on the resource, the only way to complete the proof is to use a corresponding *closing view shift*, of which we have only one, to view shift the updated resource back to Q. Since this marks the point in time where the atomic action commits, we also call this the *committing view shift*. Note that since we do not know anything about R, we have no choice but to use the same M with Q that we originally got from the opening view shift.

Sometimes, however, one has to deal with an operation that *could* be the linearization point, but it is not known up front (*i.e.,* until reasoning about the postcondition) whether that will be the case. For this reason, it is necessary to be able to *abort* an atomic update. For example, recv internally uses **tryrecv**, which has two possible outcomes: either a message was received, in which case we can commit and are done, or the channel was empty, in which case we loop again. Thus, in the proof, we would be stuck if the only closing view shift we had at our disposal was the committing one. Hence we extend (7) to assume an additional *aborting view shift* (as another closing view shift), which lets us go back to P if the commit did not happen. We arrive at the following, general pattern:

$$\begin{aligned}&\forall P, Q, R, \mathcal{E}'.\ (P\ {}^{\mathcal{E} \uplus \mathcal{E}'}\!\!\Leftrightarrow^{\mathcal{E}}\ \exists x.\ \alpha(x) * R(x)) \land \\ &\quad (\forall x, v.\ \beta(x, v)\ * R(x)\ {}^{\mathcal{E}}\!\!\Rrightarrow^{\mathcal{E} \uplus \mathcal{E}'}\ Q(x, v)) \\ &\quad \Rightarrow \{P\}\ e\ \{v.\ \exists x.\ Q(x, v)\}_{\mathcal{E} \uplus \mathcal{E}'}\end{aligned} \tag{8}$$

Leaving aside the generalization of the predicates, the only difference from (7) is that the first view shift is now bidirectional.

Formal definition. We define logically atomic triples in Figure 13. Note that x is free in α, R, β and Q, while v is free in β and Q, and also note that $\mathcal{E}_1 \# \mathcal{E}_2$ denotes disjointness of invariant masks. The definition differs from (8) only in a few technical details.

First of all, we collect the antecedent of the implication (8) into its own syntactic sugar, which we call an *atomic shift*. An atomic triple is, roughly, an implication from an atomic shift to a normal Hoare triple that ties the pre- and postconditions of the triple and the shift together. We give this Hoare triple a fixed mask, $\top$. Since e will generally not be physically atomic, ACSQ cannot be applied, so a smaller mask would be of no use to a client.

The definition mentions three masks: $\mathcal{E}$, $\mathcal{E}_M$, and $\mathcal{E}_R$. What are they? Well, it turns out that rather than talking about the invariants that the client *may* rely on being enabled (that would be $\mathcal{E} \uplus \mathcal{E}'$ in (8)), it is often more convenient to talk about those invariants the client *may not* rely on. (In Figure 13, we write $-\mathcal{E}$ to denote $\top \setminus \mathcal{E}$.)

LAAtomic
$$\frac{e \text{ phys. atomic} \quad \forall x.\ \{\alpha\}\ e\ \{v.\ \beta\}_{\mathcal{E}_M}}{\langle x.\ \alpha\rangle\ e\ \langle v.\ \beta\rangle^{\mathcal{E}_M}}$$

LAHoare
$$\frac{\langle x.\ \alpha\rangle\ e\ \langle v.\ \beta\rangle^{\mathcal{E}_M}_{\mathcal{E}} \qquad \forall x.\ \mathsf{timeless}(\alpha)}{\forall x.\ \{\alpha\}\ e\ \{v.\ \beta\}_{\top}}$$

LAFrame
$$\frac{\langle x.\ \alpha\rangle\ e\ \langle v.\ \beta\rangle^{\mathcal{E}_M}_{\mathcal{E}} \qquad \mathcal{E}' \mathrel{\#} \mathcal{E}_M}{\langle x.\ \alpha * P\rangle\ e\ \langle v.\ \beta * P\rangle^{\mathcal{E}_M}_{\mathcal{E}\uplus\mathcal{E}'}}$$

LAExist
$$\frac{\langle x, y.\ \alpha\rangle\ e\ \langle v.\ \beta\rangle^{\mathcal{E}_M}_{\mathcal{E}}}{\langle x.\ \exists y.\ \alpha\rangle\ e\ \langle v.\ \beta\rangle^{\mathcal{E}_M}_{\mathcal{E}}}$$

LACsq
$$\frac{\forall x.\ \alpha\ {}^{\mathcal{E}\uplus\mathcal{E}'}{\Longleftrightarrow}^{\mathcal{E}}\ \alpha' \qquad \langle x.\ \alpha'\rangle\ e\ \langle v.\ \beta'\rangle^{\mathcal{E}_M}_{\mathcal{E}} \qquad \forall x, v.\ \beta'\ {}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}\uplus\mathcal{E}'}\ \beta \qquad \mathcal{E}' \mathrel{\#} \mathcal{E}_M}{\langle x.\ \alpha\rangle\ e\ \langle v.\ \beta\rangle^{\mathcal{E}_M}_{\mathcal{E}\uplus\mathcal{E}'}}$$

LAInv
$$\frac{\langle x.\ \triangleright R * \alpha\rangle\ e\ \langle v.\ \triangleright R * \beta\rangle^{\mathcal{E}_M}_{\mathcal{E}} \qquad \iota \notin \mathcal{E}_M}{\boxed{R}^{\iota} \vdash \langle x.\ \alpha\rangle\ e\ \langle v.\ \beta\rangle^{\mathcal{E}_M}_{\mathcal{E}\uplus\{\iota\}}}$$

LASts
$$\frac{\langle x, s.\ s \in \uparrow(\{s_0\}, T) * \triangleright\varphi(s) * \alpha\rangle\ e\ \langle v.\ \exists s', T'.\ (s, T) \rightarrow^{*} (s', T') * \triangleright\varphi(s') * \beta\rangle^{\mathcal{E}_M}_{\mathcal{E}} \qquad \iota \notin \mathcal{E}_M}{\boxed{\mathsf{STSInv}(\mathcal{S}, \varphi, \gamma)}^{\iota} \vdash \left\langle x.\ \boxed{(s_0, T) : \textsc{Sts}_{\mathcal{S}}}^{\gamma} * \alpha\right\rangle\ e\ \left\langle v.\ \exists s', T'.\ \boxed{(s', T') : \textsc{Sts}_{\mathcal{S}}}^{\gamma} * \beta\right\rangle^{\mathcal{E}_M}_{\mathcal{E}\uplus\{\iota\}}}$$

Figure 14. Selected proof rules for logically atomic triples.

Context: $\left\langle \begin{array}{l} M.\ P \Longleftrightarrow c \prec M \mid R, \mathcal{E}_R \mid \\ m.\ c \prec M \setminus \{m\} \wedge m \in M \Rrightarrow Q(M, m) \end{array} \right\rangle^{\mathcal{E}_{\mathsf{chan}}}_{\emptyset}$

$\{P\}_{\top}$

LAHoare

$\quad \langle P\rangle_{-\mathcal{E}_{\mathsf{chan}}}$

$\quad$ Open $\mathcal{E}_R$

$\qquad \langle c \prec M * R(M)\rangle_{-\mathcal{E}_{\mathsf{chan}}, \mathcal{E}_R}$

$\qquad$ **let** $v =$ **tryrecv** c **in**

$$\left\langle \begin{array}{l} \big(c \prec \emptyset \wedge v = \mathsf{None} \wedge M = \emptyset \vee \\ \exists m.\ c \prec M \setminus \{m\} \wedge m \in M \wedge \\ v = \mathsf{Some}(m)\big) * R(M) \end{array} \right\rangle_{-\mathcal{E}_{\mathsf{chan}}, \mathcal{E}_R}$$

$$\left\langle \begin{array}{l} P * v = \mathsf{None} \vee \\ \exists M, m.\ Q(M, m) * v = \mathsf{Some}(m) \end{array} \right\rangle_{-\mathcal{E}_{\mathsf{chan}}}$$

$\{P * v = \mathsf{None} \vee \exists M, m.\ Q(M, m) * v = \mathsf{Some}(m)\}_{\top}$

case v **of** None $\Rightarrow$ $\{P\}_{\top}$ *recv* c $\{m.\ \exists M.\ Q(M, m)\}_{\top}$

$\mid$ Some(m) $\Rightarrow$ $\{\exists M, v.\ Q(M, m)\}_{\top}$ m $\{m.\ \exists M.\ Q(M, m)\}_{\top}$

Figure 15. Proof outline for recv.

The first two masks describe invariants that the module knows and cares about. We call $\mathcal{E}_M$ the *module mask*: these are the invariants the module wants to be able to open up before calling client view shifts, so it is essential that the client *not* depend on these invariants at all. We call $\mathcal{E}$ the *shared mask*: these are the invariants that both the module and the client may depend on, and that therefore the client may not disable in its atomic shift. Both of these masks show up in logically atomic triples themselves.

The third mask, $\mathcal{E}_R$, we call the *client mask*. This is a set of invariants that the client depends on *and* disables in its atomic shift. The module does not know or care about these invariants, and thus the client can choose $\mathcal{E}_R$ arbitrarily, so long as it is disjoint from the first two masks.

The atomic shift imposes a (somewhat odd) restriction that P must be timeless, the reason for which is as follows. It is necessary, when opening an invariant, to surround it with the $\triangleright$ modality (see InvOpen). But with logically atomic triples, it is often the case that invariants are opened in a nested fashion, *i.e.,* the caller opens an invariant "around" the triple (using LAInv, Figure 14), and the module itself opens another invariant at the linearization point. Ultimately, however, this all happens in an application of ACsq around a single, physical step of the underlying language. So there is just one physical step available to strip off a single $\triangleright$. This issue is solved by making P timeless, which means we can use VSTimeless to deal with the $\triangleright$. In practice, this is not a serious restriction: The assertions we use are typically either timeless (like ghost resources) or pure (like invariant assertions). It is thus possible to move the pure assertions into the context (which means we can make them available anywhere), and to put the (timeless) resources into P.

Finally, note that atomic shifts and atomic triples (like view shifts and Hoare triples) are pure facts and can therefore be duplicated.

Derived rules. These definitions support the derived rules given in Figure 14. Most important is the rule of consequence (LACsq), which (like the original ACsq) may be applied to view shifts that change masks. But with LACsq, we can open invariants around expressions like recv that are not physically atomic! In other words, for any expression satisfying a logically atomic specification, we can apply well-established reasoning patterns (*e.g.,* LAInv, LASts) that, in most logics, work only for physically atomic expressions.

LAAtomic shows that any physically atomic triple is also logically atomic. The rule LAExist permits us to bind a variable in the atomic shift. Unlike in the standard existential rule for Hoare triples (cf. our appendix [1]), it would be unsound here to move y all the way out of the triple to a universal quantifier. Finally, LAHoare can be used to convert logically atomic triples to normal ones. This is where the timeless restriction rears its (only slightly ugly) head.

7.2 Proof of blocking receive

As an example of working with logically atomic specifications, we are going to derive the already mentioned specification for recv:

$$\forall c.\ \langle M.\ c \prec M\rangle\ \mathsf{recv}\ c\ \langle m.\ c \prec M \setminus \{m\} \wedge m \in M\rangle^{\mathcal{E}_{\mathsf{chan}}} \qquad (9)$$

First, we unfold the syntactic sugar for atomic triples. We thus get to assume some P, Q, R, $\mathcal{E}_R$ such that the atomic shift shown at the top of Figure 15 holds.

The most important step of the proof (outlined in Figure 15) is to establish the following triple:

$$\langle P\rangle \quad \textbf{tryrecv}\ c \qquad (10)$$
$$\left\langle \begin{array}{l} v.\ P * v = \mathsf{None} \vee \\ \quad \exists M, m.\ Q(M, m) * v = \mathsf{Some}(m) \end{array} \right\rangle^{\mathcal{E}_{\mathsf{chan}}}_{-\mathcal{E}_{\mathsf{chan}}}$$

Intuitively, **tryrecv** either returns some message and completes the update (yielding Q), or it returns None and maintains P.

Once we have shown this, we can use LAHoare to obtain a normal Hoare triple, and complete the proof with standard reasoning.

To show (10), we will have to use the atomic shift. It is clear what to do in the precondition: we are going to open the invariants $\mathcal{E}_R$ with the opening view shift, obtaining $c \prec M$. For the postcondition, we are going to use the committing view shift in case **tryrecv** succeeded, and the aborting view shift otherwise. Hence we have to show:

$$\langle \exists M.\ c \prec M * R(M)\rangle \quad \textsf{tryrecv}\ c \quad \left\langle \begin{array}{l} v.\ c \prec \emptyset * R(\emptyset) * v = \textsf{None} \lor \\ \quad \exists M, m.\ c \prec M \setminus \{m\} \land m \in M * \\ \qquad R(M) * v = \textsf{Some}(m) \end{array} \right\rangle^{\mathcal{E}_{\textsf{chan}}}_{-\mathcal{E}_{\textsf{chan}}, \mathcal{E}_R} \tag{11}$$

Now we can apply LAExist to bind the M, followed by LACsq (but without changing masks) to make use of this bound variable in the postcondition.

$$\langle M.\ c \prec M * R(M)\rangle \quad \textsf{tryrecv}\ c \quad \left\langle \begin{array}{l} v.\ \big(c \prec \emptyset \land v = \textsf{None} \land M = \emptyset \lor \\ \quad \exists m.\ c \prec M \setminus \{m\} \land m \in M \land \\ \qquad v = \textsf{Some}(m)\big) * R(M) \end{array} \right\rangle^{\mathcal{E}_{\textsf{chan}}}_{-\mathcal{E}_{\textsf{chan}}, \mathcal{E}_R} \tag{12}$$

Note that after framing $R(M)$ with LAFrame, this is exactly a logically atomic version of the spec for **tryrecv** given in Figure 11. We are thus just a single application of LAAtomic away from completing the proof. This also shows that our construction only requires a logically atomic spec for **tryrecv**—it does not depend on **tryrecv** being physically atomic.

Together with the triples for **send** and **newch** in Figure 11 and an application of LAAtomic, this completes the proof of the channel specification from Figure 12. With this example, we have illustrated (1) how to prove a logically atomic specification for recv, and (2) how the logically atomic specification of **tryrecv** supports reasoning principles normally reserved for physically atomic expressions.

8. Putting logical atomicity to work

Now that we have proven a logically atomic spec for recv, we are in a position to state and prove a logically atomic spec for mutable references, using the well-known encoding [26] in Figure 16.

The idea behind this encoding is to represent a reference as a channel, on which a background "server" process listens for requests. Allocating a reference (ref e) involves allocating a fresh channel r (which represents the reference) and forking off a server process (srv r e). This server process will listen for messages from client processes that send it Get, Set, and Cas requests on r in order to perform reads and writes on the reference. Each such request includes a freshly generated reply channel d, along which the server sends the result of the requested operation.

By analogy with ML-style modules, we wish to show that this encoding is a "functor":[2] If $e_{\textsf{newch}}$, $e_{\textsf{send}}$, and $e_{\textsf{recv}}$ satisfy a logically atomic spec for channels, then ref, !, :=, and cas satisfy such a spec for references.

In Figures 17 and 18, we define predicates $\varphi_{\textsf{chan}}$ and $\varphi_{\textsf{ref}}$ to represent the "signatures" of channels and references, in which the specific masks, expressions, and abstract predicates (that a particular implementation would define) are held abstract. The concrete predicate $\prec$ in Figure 12 satisfies these properties: we can implement $\varphi_{\textsf{chan}}$. Note that we require the abstract predicates to be timeless so they can be used with LAHoare.

In our appendix [1], we verify references as channels by proving

$$\begin{array}{l} \forall \mathcal{E}_{\textsf{chan}}, \mathcal{E}_{\textsf{ref}}, e_{\textsf{newch}}, e_{\textsf{send}}, e_{\textsf{recv}}. \\ \quad \varphi_{\textsf{chan}}(\mathcal{E}_{\textsf{chan}}, e_{\textsf{newch}}, e_{\textsf{send}}, e_{\textsf{recv}}) \land \\ \quad \mathcal{E}_{\textsf{chan}} \subseteq \mathcal{E}_{\textsf{ref}} \land \textsf{infinite}(\mathcal{E}_{\textsf{ref}} \setminus \mathcal{E}_{\textsf{chan}}) \\ \quad \Rrightarrow_{\mathcal{E}_{\textsf{ref}}} \varphi_{\textsf{ref}}(\mathcal{E}_{\textsf{ref}}, \textsf{ref}, !, :=, \textsf{cas}) \end{array} \tag{13}$$

[2] To our knowledge, the correctness of this encoding has not been specified and proven in this modular style before. Prior work [38] has shown that an entire language with references can be faithfully translated into π-calculus.

Let expressions $e_{\textsf{newch}}$, $e_{\textsf{send}}$, and $e_{\textsf{recv}}$ be given. Define

$$\begin{aligned} \textsf{ref}\ e &\triangleq \textbf{let}\ r = e_{\textsf{newch}}\ \textbf{in fork}\ \textsf{srv}\ r\ e; r \\ !e &\triangleq \textsf{rpc}\ e\ \textsf{Get} \\ e := e' &\triangleq \textsf{rpc}\ e\ \textsf{Set}(e') \\ \textsf{cas}(e, e_1, e_2) &\triangleq \textsf{rpc}\ e\ \textsf{Cas}(e_1, e_2) \end{aligned}$$

where

$$\begin{array}{l} \textsf{rpc} \triangleq \lambda r.\ \lambda m. \\ \qquad \textbf{let}\ d = e_{\textsf{newch}}\ \textbf{in} \\ \qquad e_{\textsf{send}}(r, (d, m)); e_{\textsf{recv}}\ d \\ \textsf{srv} \triangleq \lambda r.\ \textbf{rec}\ \mathit{loop}(v). \\ \qquad \textbf{let}\ (d, m) = e_{\textsf{recv}}\ r\ \textbf{in} \\ \qquad \textbf{let}\ \mathit{reply} = \lambda m'.\ \lambda v'.\ (e_{\textsf{send}}(d, m'); \mathit{loop}\ v')\ \textbf{in} \\ \qquad \textbf{case}\ m\ \textbf{of} \\ \qquad\qquad \textsf{Get} \Rightarrow \mathit{reply}\ v\ v \\ \qquad\quad \mid \textsf{Set}(w) \Rightarrow \mathit{reply}\ ()\ w \\ \qquad\quad \mid \textsf{Cas}(v_1, v_2) \Rightarrow \textbf{let}\ b = (v = v_1)\ \textbf{in} \\ \qquad\qquad\qquad\qquad\quad \textbf{let}\ v' = \textbf{if}\ b\ \textbf{then}\ v_2\ \textbf{else}\ v\ \textbf{in} \\ \qquad\qquad\qquad\qquad\quad \mathit{reply}\ b\ v' \end{array}$$

Figure 16. Implementing references as channels.

Since the reference operations use the channel interface, we require that all invariants needed by the channel operations are available in $\mathcal{E}_{\textsf{ref}}$. Furthermore, the reference module allocates an invariant in order to enforce a protocol on the ghost state underlying the abstract points-to predicate. Thus there need to be infinitely many invariant names it can use for its own purposes ($\mathcal{E}_{\textsf{ref}} \setminus \mathcal{E}_{\textsf{chan}}$) to satisfy NewInv. This is also the reason that (13) is a view shift rather than an implication. Note that we can completely abstract away from how the channel operations treat their invariant names, just as users of $\varphi_{\textsf{ref}}$ do not have to care about the fact that some of the invariants in $\mathcal{E}_{\textsf{ref}}$ actually belong to the channel module. This shows that, with some small effort, proper abstraction is possible in Iris despite our use of global invariant masks at view shifts and Hoare triples.

In our proof of (13), the server thread for a reference is the one that actually commits all operations on it by performing logical updates on behalf of client requests. Hence, a client thread must transfer its atomic shift and precondition P to the server, which it does using the invariant of the reference module. (This transfer is made possible by the fact that invariants in Iris are impredicative, as explained at the end of §4.) The server can then apply the client's opening and committing view shift, sending Q back to the client. This is a particularly simple example of *helping* [35, 33], a common phenomenon in fine-grained concurrent algorithms.

To make the proof work out, we had to change the code in Figure 16 slightly: at some places, we introduced **skip** to be able to strip off a $\triangleright$. Since InvOpen adds a $\triangleright$ to the invariant, this means we have to take a physical step if (1) the invariant is opened around the last operation a function is performing and (2) the function's postcondition does not specify a $\triangleright$. The need to insert such **skip**s is a known irritation in the world of step-indexed logics, but it is perfectly sound: adding **skip**s does not affect the observable behavior of a program for the kind of observations we are considering.

As logically atomic triples are compatible with advanced techniques for reasoning with shared state (*e.g.,* LASts), it is perfectly straightforward to verify clients of the references-as-channels module against the abstract spec $\varphi_{\textsf{ref}}$. To drive this point home, our appendix [1] includes a verification of elimination stacks [16, 35],

$$
\begin{aligned}
&\varphi_{\mathsf{chan}}(\mathcal{E}_{\mathsf{chan}}, e_{\mathsf{newch}}, e_{\mathsf{send}}, e_{\mathsf{recv}}) \triangleq \Box\exists(\cdot \prec \cdot) \in \mathsf{Val} \times \mathsf{Bag} \to \mathsf{Prop}.\\
&\quad \forall c, M.\ \mathsf{timeless}(c \prec M) \wedge\\
&\quad \forall c, M, M'.\ c \prec M * c \prec M' \Rightarrow \mathsf{False} \wedge\\
&\quad \{\mathsf{True}\}\ e_{\mathsf{newch}}\ \{c.\ c \prec \emptyset\} \wedge\\
&\quad \forall c, m.\ \langle M.\ c \prec M\rangle\ e_{\mathsf{send}}(c, m)\ \langle x.\ c \prec M \uplus \{m\} \wedge x = ()\rangle^{\mathcal{E}_{\mathsf{chan}}}\\
&\quad \wedge\ \forall c.\ \langle M.\ c \prec M\rangle\ e_{\mathsf{recv}}\ c\ \langle m.\ c \prec M \setminus \{m\} \wedge m \in M\rangle^{\mathcal{E}_{\mathsf{chan}}}
\end{aligned}
$$

Figure 17. Specification for channels with blocking receive.

$$
\begin{aligned}
&\varphi_{\mathsf{ref}}(\mathcal{E}_{\mathsf{ref}}, e_{\mathsf{ref}}, e_{\mathsf{get}}, e_{\mathsf{set}}, e_{\mathsf{cas}}) \triangleq \Box\exists(\cdot \mapsto \cdot) \in \mathsf{Val} \times \mathsf{Val} \to \mathsf{Prop}.\\
&\quad \forall r, v.\ \mathsf{timeless}(r \mapsto v) \wedge\\
&\quad \forall r, v, w.\ r \mapsto v * r \mapsto w \Rightarrow \mathsf{False} \wedge\\
&\quad \forall v.\ \{\mathsf{True}\}\ e_{\mathsf{ref}}\ v\ \{r.\ r \mapsto v\} \wedge\\
&\quad \forall r.\ \langle v.\ r \mapsto v\rangle\ e_{\mathsf{get}}\ r\ \langle x.\ r \mapsto v \wedge x = v\rangle^{\mathcal{E}_{\mathsf{ref}}} \wedge\\
&\quad \forall r, v.\ \langle r \mapsto \_\rangle\ e_{\mathsf{set}}\ r\ v\ \langle x.\ r \mapsto v \wedge x = ()\rangle^{\mathcal{E}_{\mathsf{ref}}} \wedge\\
&\quad \forall r, v_1, v_2.\ \langle v.\ r \mapsto v\rangle \quad e_{\mathsf{cas}}(r, v_1, v_2)\\
&\qquad\qquad \left\langle \begin{aligned} b.\ & b = \mathsf{true} \wedge v = v_1 \wedge r \mapsto v_2 \vee\\ & b = \mathsf{false} \wedge v \neq v_1 \wedge r \mapsto v \end{aligned} \right\rangle^{\mathcal{E}_{\mathsf{ref}}}
\end{aligned}
$$

Figure 18. Specification for references.

a challenging example from the literature that employs helping in a more complex way than the references-as-channels module does. This verification is parameterized by an arbitrary implementation of references that is assumed to satisfy the logically atomic spec φ_{ref}. The code and proof closely follow those in the technical appendix of iCAP [33], the chief difference being that the iCAP proof depends on the reference operations being physically atomic, whereas ours does not. With this, we complete our stack of abstractions (Figure 1).

9. Related work

In the introduction, we motivated Iris in relation to the state of the art in modular concurrency verification. Here, we give a few more detailed comparisons with the most closely related work.

Monoids. The centrality of partial commutative monoids dates back to the earliest models of separation logic, but only recently did a number of different models and logics begin to employ PCMs as a way of characterizing more fine-grained notions of interference [22, 9, 24]. The Views framework [9], in particular, enables the user to tie logical resources (represented as a PCM of the user's choice) to physical resources, according to a particular user-defined interpretation of logical resources as assertions about physical ones. In this way, Views, like Iris, supports protocol-based reasoning.

However, Views is limited in that it effectively requires the user to bake in a fixed invariant tying logical to *physical* resources, with no logical support for layering further invariants on top of those logical resources. Iris, in contrast, provides built-in logical support for user-defined invariants over *logical* (ghost) resources; these are crucial, *e.g.,* for enabling the verification of elimination stacks to impose its own invariants over the *logical* points-to assertions exported by the references-as-channels module one layer below. Furthermore, although we have not emphasized it in this paper, Iris invariants can be defined using higher-order, guarded recursive predicates akin to [33] (unlike in Views, which only supported first-order logics).

View shifts were introduced by the Views framework as a specialization of the notion of "repartitioning" from CAP [10]. A repartitioning describes an update of ghost state in terms of an update of the underlying physical state. A view shift in [9] is thus simply a repartitioning that preserves the underlying physical state. CAP uses a two-level approach for reasoning about atomic expressions: one has to prove how the expression updates the physical state and separately prove a repartitioning that relates this back to the abstract level. In Iris, rule ACSQ offers more fine-grained control. Instead of going all the way to the concrete physical state, it is possible to open just as many levels of abstraction as are needed to justify the execution of the atomic expression.

Although we believe our monoid encoding of STSs with tokens is novel, most of the monoid constructions we used have been described previously. In particular, Dockins *et al.* [11] treat monoids in an algebraic manner, as separation algebras (SA). They provide various generic building blocks like discrete SAs (equivalent to our exclusive monoid), products and sums, as well as a generalization of fractional monoids they call *shares*. To facilitate the treatment of sums, their monoids can have multiple units. We believe that our restriction to a single unit (and a single zero) can be lifted easily. Unlike SAs, our monoids do not generally have to be cancellative.

Invariants. The idea of relating logical resources with physical resources through invariants is also present in Pilkiewicz and Pottier's work on monotonic state [30], Jensen and Birkedal's work on "fictional separation logic" [20], and Krishnaswami *et al.*'s work on "superficially substructural types" [22]. They are all restricted to a sequential setting. Pilkiewicz and Pottier's type-and-capability system uses "fates"—ghost variables whose values grow monotonically—to reason locally about lower bounds on ghost state. Using invariants, they lift these reasoning principles to monotonic physical state. Invariants and fates are orthogonal, independent principles. Iris supports similar reasoning principles for monotonic state, through an encoding of fates as STSs. Fictional separation logic (FSL) combines monoids with indirect Hoare triples specifying an interpretation map, which serves as invariant. Their approach is based on a syntactic translation into a standard separation logic and it is not clear how this approach extends to a concurrent setting. Superficially substructural types (SST) combines the approach of a type system with the more flexible forms of sharing supported by arbitrary monoids in FSL. As we have shown in §5.2 (rule AUTH), we can support reasoning in the style of SST's "sharing rule".

In the concurrent setting, previous work—*e.g.,* on CaReSL [34], iCAP [33], and TaDA [8]—has typically fixed a particular and very useful monoid construction. Iris can be used to encode the patterns of reasoning found in these logics, but with a simpler set of primitive mechanisms and proof rules (whose soundness we have verified in Coq [1]). In addition, on a technical level, Iris is more general and expressive than these earlier logics. Like iCAP but unlike CaReSL and TaDA, Iris supports full higher-order logic and impredicative invariants, which are useful for giving modular specifications of libraries. Going beyond iCAP, Iris supports mask-changing view shifts; these are essential to our ability to encode TaDA-style logically atomic specs, as shown in §7. The notions of logical atomicity developed in TaDA and Iris are closely related, as is evident by the extra binder both had to introduce (the "funny" universal quantifier in TaDA, and the binder in the precondition of Iris's logically atomic triples), as well the similarity of the proof rules. At this point, Iris lacks the "private" pre- and postcondition of TaDA, but they would be easy to add to the syntactic sugar.

Like Iris, Cohen *et al.* [7] strive to provide a minimal basis for concurrent reasoning, but theirs is based on a ghost heap and two-state invariants, whereas ours is based on arbitrary ghost PCMs and one-state invariants. The two approaches are optimizing for different goals. Our logic is substructural and supports more expressive—*e.g.,* higher-order and logically atomic—specifications. Theirs is not, making it better suited to use with automated verification tools like

SMT solvers. However, more work remains to be done to sort out the precise formal relationship between the two orthogonal bases.

Atomicity. Concerning the proving of logically atomic specs, TaDA is currently not able to reason about fine-grained data structures that employ inter-thread cooperation (aka "helping"). In Iris, this is possible thanks to the impredicative invariants (see the example in §8), similar to the approach taken in iCAP, which in turn was inspired by previous work of Jacobs and Piessens [19] on VeriFast. To support impredicativity and thus also helping, our model of Iris is based on a solution to a recursive domain equation. That is avoided in VeriFast, which instead attempts to use a kind of Gödel encoding of predicates. However, according to Jacobs [18], the soundness and the generality of this encoding approach of VeriFast are unclear.

Liang and Feng [25] propose another approach to observable atomicity, based on linearizability. Like CaReSL, they appeal to an extra-logical theorem when verifying clients against modules with observably atomic behavior. However, following ideas from Turon *et al.* [35], their logic also supports *speculative* execution of linearization points, which they use to establish soundness of the reasoning patterns developed by Vafeiadis in his thesis [36].

Further work remains to be done in order to extend Iris with support for such speculative reasoning [2]. (The same goes for iCAP and TaDA as well.) Speculation is important in verifying fine-grained concurrent ADTs in which the ordering of a sequence of linearization points may not be known "in real time" but only after the corresponding operations have completed. Intuitively, speculation is challenging in Iris because, when we verify that a module satisfies a logically atomic spec, we perform updates to the state of the module using view shifts provided by the client, so the client *can* (in theory) observe the linearization points in real time. To address this problem, a natural starting point would be to try to follow the model of speculation in Turon *et al.* [35] and generalize Iris assertions from predicates over resources to predicates over *sets* of resources, but it is not yet clear how this would fit into our general approach to logical atomicity.

Acknowledgments

This research is supported in part by a Microsoft Research PhD Scholarship, and in part by the ModuRes Sapere Aude Advanced Grant from The Danish Council for Independent Research for the Natural Sciences (FNU). We thank Lisa Kohl for fruitful discussions during her internship at MPI-SWS, and we also thank the anonymous reviewers for extremely helpful and detailed feedback.

References

[1] Appendix and Coq development. `http://plv.mpi-sws.org/iris`.

[2] M. Abadi and L. Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.

[3] A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. In *POPL*, 2009.

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

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

[6] L. Birkedal, B. Reus, J. Schwinghammer, K. Støvring, J. Thamsborg, and H. Yang. Step-indexed Kripke models over recursive worlds. In *POPL*, 2011.

[7] E. Cohen, *et al.* Invariants, modularity, and rights. In *PSI*, 2009.

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

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

[10] T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP*, 2010.

[11] R. Dockins, A. Hobor, and A. W. Appel. A fresh look at separation algebras and share accounting. In *APLAS*, 2009.

[12] X. Feng. Local rely-guarantee reasoning. In *POPL*, 2009.

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

[14] I. Filipović, P. O'Hearn, N. Torp-Smith, and H. Yang. Blaming the client: On data refinement in the presence of pointers. In *FACS*, 2009.

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

[16] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *SPAA*, 2004.

[17] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12(3):463–492, 1990.

[18] B. Jacobs. Personal communication, 2014.

[19] B. Jacobs and F. Piessens. Expressive modular fine-grained concurrency specification. In *POPL*, 2011.

[20] J. B. Jensen and L. Birkedal. Fictional separation logic. In *ESOP*, 2012.

[21] C. B. Jones. Tentative steps toward a development method for interfering programs. *TOPLAS*, 5(4):596–619, 1983.

[22] N. R. Krishnaswami, A. Turon, D. Dreyer, and D. Garg. Superficially substructural types. In *ICFP*, 2012.

[23] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, 1979.

[24] R. Ley-Wild and A. Nanevski. Subjective auxiliary state for coarse-grained concurrency. In *POPL*, 2013.

[25] H. Liang and X. Feng. Modular verification of linearizability with non-fixed linearization points. In *PLDI*, 2013.

[26] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[27] A. Nanevski, R. Ley-Wild, I. Sergey, and G. A. Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *ESOP*, 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] A. Pilkiewicz and F. Pottier. The essence of monotonic state. In *TLDI*, 2011.

[31] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.

[32] F. Sieczkowski, A. Bizjak, Y. Zakowski, and L. Birkedal. Modular reasoning about concurrent higher-order imperative programs: a Coq tutorial. `http://users-cs.au.dk/birke/modures/tutorial/index.html`, 2014.

[33] K. Svendsen and L. Birkedal. Impredicative concurrent abstract predicates. In *ESOP*, 2014.

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

[35] A. Turon, J. Thamsborg, A. Ahmed, L. Birkedal, and D. Dreyer. Logical relations for fine-grained concurrency. In *POPL*, 2013.

[36] V. Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2007.

[37] V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, 2007.

[38] D. Walker. Objects in the pi-calculus. *Inf. Comput.*, 116(2):253–271, 1995.